

前 言

本书第 1 版在 2008 年初出版以后,受到广大读者的支持和厚爱,累计销售 1.6 万册,从几年的市场和读者反馈看,在第 1 版中还存在一些不足,主要是以下几方面。

- 没有现成的开发环境,读者需要从头到尾构建,而构建需要花费很长的时间,许多时候会不成功,加之配套光盘中的实例没有 Makefile,更加大了操作的难度。
- 没有配套的开发板,大量的基于 S3C2410 的实例读者身边如果没有可以直接运行的平台,就无法亲身体验这些驱动。
- 个别内容实用性不强或过于陈旧,也有个别知识点的讲解语言晦涩,读者不易理解,如 platform 驱动。
- 一些知识点内容不够完整,如 input 驱动、USB UDC 和 gadget 驱动、SPI 驱动、ASoC 驱动等。

鉴于此,作者针对以上问题对第 1 版内容进行修订,推出了第 2 版。新版中对全书超过 40% 的内容进行了修订。这些修订,一些是对过时内容的删除,一些是对讲解不清的知识点的修正,一些是对重点内容的增强,一些则是有用知识点的增加,其目的是为读者提供一套更加准确和完整的全方位、立体式 Linux 设备驱动学习平台。第 2 版相对第 1 版的主要改动如下。

(1) 直接提供 VirtualBox 虚拟机,该虚拟机上已包含了书中所需的开发环境和源代码,读者不再需要安装环境即可进行实验,书中详细介绍了各个实验的步骤。

(2) 提供了专门的配套学习板——基于三星 S3C6410 SoC 的 LDD6410 (Linux Device Drivers 6410),使得书中的各种真实设备驱动实例有了实验的依托。

(3) 全面升级内核至 Linux 2.6.28.6,根据 Linux 内核 API 的变更情况更新了书中的所有内容,如 I²C 驱动的体系结构、网络 NAPI 的接口等,并对 delayed_work 等较新的内核机制进行了介绍。

(4) 删除了过时的内容,如传统的按键驱动、SAA7113H 启动、传统的 IDE 驱动等,同时新增了大量内容,包括 Linux 内核的编码风格、Linux 内核的移植、Android 驱动、USB UDC 和 gadget 驱动、ALSA SoC 驱动、input 驱动、SPI 驱动、基于 sysfs 的设备驱动、Linux 设备驱动的固件加载、Linux 性能调优工具、Linux 设备驱动的电源管理、Linux 驱动的分层设计思想、主机驱动与设备驱动分离设计思想等。

(5) 在块设备驱动方面,删除了 RAMDISK 驱动实例,而新增了更加简单易懂的 vmem_disk、类似于 globalmem 和 globalfifo 驱动。

(6) 对许多关键知识点的讲解进行了语言调整和内容增强,以便读者能更好地理解,例如,以专门章节讲解 platform 驱动等。

全书总体结构仍然与第 1 版一致,共分 4 篇 23 章,内容安排如下。

第 1 篇(第 1~3 章)主要讲解 Linux 设备驱动的基础。

第 1 章主要讲解设备驱动的作用,并从无操作系统的设备驱动引出了 Linux 操作系统下的设



备驱动以及全书所用实验环境的安装方法。

第 2 章系统地讲解了一个 Linux 驱动工程师应该掌握的硬件知识,使读者打下 Linux 设备驱动开发的硬件基础。本章涵盖了各种类型的 CPU、存储器和常见的外设,并讲解了硬件时序分析方法和仪器使用方法。

第 3 章将 Linux 设备驱动放在 Linux 2.6 内核背景中进行讲解,说明 Linux 内核的基本原理和编程方法,为编写 Linux 设备驱动打下软件基础。

第 2 篇(第 4~12 章)主要讲解 Linux 设备驱动编程的基础理论、字符设备驱动、设备驱动设计中涉及的并发控制、同步等问题以及 Linux 驱动的工程化。

第 4、5 章分别讲解 Linux 内核模块和 Linux 设备文件系统。

第 6~9 章以虚拟设备 globalmem 和 globalfifo 为主线讲解了字符设备驱动的编写方法,并讲解了并发控制、阻塞与非阻塞、异步 I/O 等高级控制功能。

第 10、11 章分别讲解 Linux 驱动编程中所涉及的中断和定时器,内核和 I/O 操作处理方法。

globalmem 和 globalfifo 驱动与真实项目中看到的驱动有一些不同,第 12 章详细讲解 Linux 设备驱动的工程化问题,让读者了解真实的驱动要考虑的诸多问题。

第 3 篇(第 13~21 章)深刻剖析复杂设备驱动的体系架构,每一章都给出了具体的实例,涉及的设备包括块设备、终端设备、I²C 适配器与 I²C 设备、网络设备、PCI 设备、USB 主机控制器、USB 设备、UDC、gadget、LCD 设备、Flash 设备等。本篇的讲解抽象与具体相结合,先以模板的形式给出各种设备驱动的设计框架,然后用具体实例设备的驱动填充对应的模板。

第 4 篇(第 22~23 章)详细讲解了 Linux 设备驱动和内核的调试和移植方法。

第 22 章讲解了 Linux 设备驱动的开发环境构建以及借助 printk、oops、/proc、strace、仿真器进行驱动调试的方法,最后介绍了 Linux 的性能调优工具。

第 23 章讲解了开发可移植驱动程序以及借助芯片范例程序、demo 板驱动和其他操作系统驱动等现成代码进行 Linux 驱动快速移植的方法,最后介绍了如何在一块新的 SoC 和电路板上构建 Linux。

本书的结构及内容参见附图。

最后,再次对广大读者以及所有为本书提出过宝贵意见、为本书的诞生奉献过力量的人们表示最诚挚的谢意!读者朋友可继续通过本书专用网址 <http://www.linuxdriver.cn> 与作者和编辑团队进行交流。

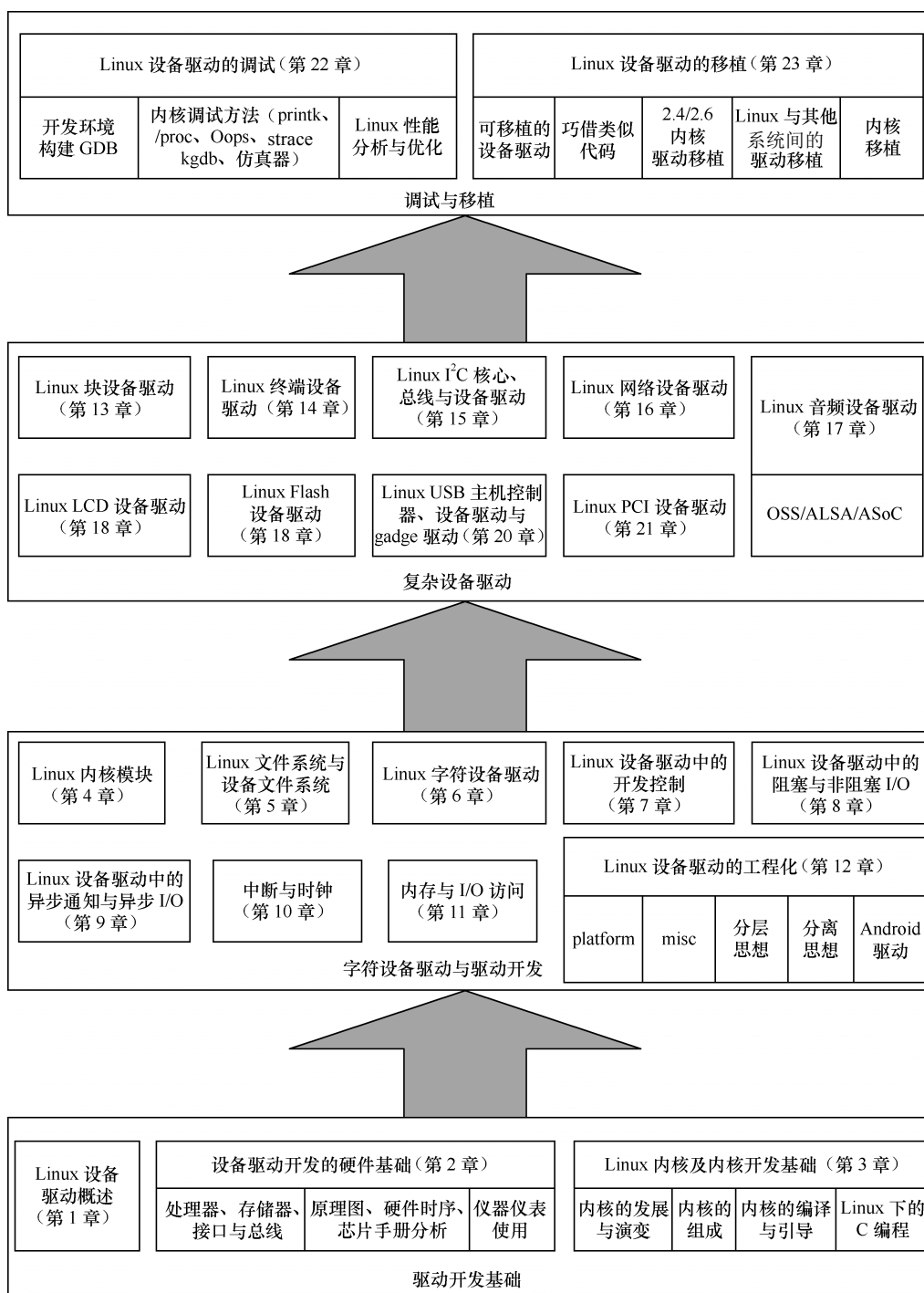
本书服务 QQ: 1275822672

服务 E-mail: book@Linuxdriver.cn

读者可以邮件咨询索取相关资料

宋宝华

2010 年 6 月于上海浦东



目 录

第 1 篇 Linux 设备驱动入门

第 1 章 Linux 设备驱动概述及开发环境构建	2
1.1 设备驱动的作用	3
1.2 无操作系统时的设备驱动	3
1.3 有操作系统时的设备驱动	5
1.4 Linux 设备驱动	6
1.4.1 设备的分类及特点	6
1.4.2 Linux 设备驱动与整个软硬件系统的关系	7
1.4.3 Linux 设备驱动的重点、难点	8
1.5 Linux 设备驱动开发环境构建	8
1.5.1 PC 上的 Linux 环境	8
1.5.2 LDD6410 开发板	11
1.5.3 工具链安装	13
1.5.4 主机端 nfs 和 tftp 服务安装	14
1.5.5 源代码阅读和编辑	14
1.6 设备驱动 Hello World: LED 驱动	15
1.6.1 无操作系统时的 LED 驱动	15
1.6.2 Linux 下的 LED 驱动	16
1.7 全书结构	19
第 2 章 驱动设计的硬件基础	21
2.1 处理器	22
2.1.1 通用处理器	22
2.1.2 数字信号处理器	23
2.2 存储器	25
2.3 接口与总线	29
2.3.1 串口	29



2.3.2	I ² C	30
2.3.3	USB	31
2.3.4	以太网接口	33
2.3.5	ISA	34
2.3.6	PCI 和 cPCI	35
2.4	CPLD 和 FPGA	38
2.5	原理图分析	41
2.5.1	原理图分析的内容	41
2.5.2	原理图的分析方法	41
2.6	硬件时序分析	43
2.6.1	时序分析的概念	43
2.6.2	典型硬件时序	44
2.7	芯片手册阅读方法	45
2.8	仪器仪表使用	48
2.8.1	万用表	48
2.8.2	示波器	48
2.8.3	逻辑分析仪	50
2.9	总结	52
第 3 章	Linux 内核及内核编程	53
3.1	Linux 内核的发展与演变	54
3.2	Linux 2.6 内核的特点	55
3.3	Linux 内核的组成	56
3.3.1	Linux 内核源代码目录结构	56
3.3.2	Linux 内核的组成部分	57
3.3.3	Linux 内核空间与用户空间	60
3.4	Linux 内核的编译及加载	60
3.4.1	Linux 内核的编译	60
3.4.2	Kconfig 和 Makefile	62
3.4.3	Linux 内核的引导	69
3.5	Linux 下的 C 编程特点	71
3.5.1	Linux 编码风格	71
3.5.2	GNU C 与 ANSI C	73
3.5.3	do { } while(0)	77

3.5.4 goto	78
3.6 总结	79

第 2 篇 Linux 设备驱动核心理论

第 4 章 Linux 内核模块	82
4.1 Linux 内核模块简介	83
4.2 Linux 内核模块程序结构	85
4.3 模块加载函数	85
4.4 模块卸载函数	86
4.5 模块参数	87
4.6 导出符号	88
4.7 模块声明与描述	89
4.8 模块的使用计数	89
4.9 模块的编译	90
4.10 使用模块绕过 GPL	91
4.11 总结	91
第 5 章 Linux 文件系统与设备文件系统	92
5.1 Linux 文件操作	93
5.1.1 文件操作系统调用	93
5.1.2 C 库文件操作	95
5.2 Linux 文件系统	97
5.2.1 Linux 文件系统目录结构	97
5.2.2 Linux 文件系统与设备驱动	98
5.3 devfs 设备文件系统	101
5.4 udev 设备文件系统	102
5.4.1 udev 与 devfs 的区别	102
5.4.2 sysfs 文件系统与 Linux 设备模型	104
5.4.3 udev 的组成	110
5.4.4 udev 规则文件	111
5.4.5 创建和配置 mdev	113
5.5 LDD6410 的 SD 和 NAND 文件系统	114



5.6 总结	117
第 6 章 字符设备驱动	118
6.1 Linux 字符设备驱动结构	119
6.1.1 cdev 结构体	119
6.1.2 分配和释放设备号	120
6.1.3 file_operations 结构体	120
6.1.4 Linux 字符设备驱动的组成	122
6.2 globalmem 虚拟设备实例描述	124
6.3 globalmem 设备驱动	125
6.3.1 头文件、宏及设备结构体	125
6.3.2 加载与卸载设备驱动	126
6.3.3 读写函数	127
6.3.4 seek 函数	128
6.3.5 ioctl 函数	129
6.3.6 使用文件私有数据	130
6.4 globalmem 驱动在用户空间的验证	136
6.5 总结	138
第 7 章 Linux 设备驱动中的并发控制	139
7.1 并发与竞态	140
7.2 中断屏蔽	141
7.3 原子操作	142
7.3.1 整型原子操作	142
7.3.2 位原子操作	142
7.4 自旋锁	143
7.4.1 自旋锁的使用	143
7.4.2 读写自旋锁	145
7.4.3 顺序锁	147
7.4.4 读-拷贝-更新	148
7.5 信号量	152
7.5.1 信号量的使用	152
7.5.2 信号量用于同步	154

7.5.3	完成量用于同步	154
7.5.4	自旋锁 vs 信号量	155
7.5.5	读写信号量	155
7.6	互斥体	156
7.7	增加并发控制后的 globalmem 驱动	157
7.8	总结	160
第 8 章	Linux 设备驱动中的阻塞与非阻塞 I/O	161
8.1	阻塞与非阻塞 I/O	162
8.1.1	等待队列	162
8.1.2	支持阻塞操作的 globalfifo 设备驱动	166
8.1.3	在用户空间验证 globalfifo 的读写	171
8.2	轮询操作	172
8.2.1	轮询的概念与作用	172
8.2.2	应用程序中的轮询编程	172
8.2.3	设备驱动中的轮询编程	172
8.3	支持轮询操作的 globalfifo 驱动	173
8.3.1	在 globalfifo 驱动中增加轮询操作	173
8.3.2	在用户空间验证 globalfifo 设备的轮询	174
8.4	总结	175
第 9 章	Linux 设备驱动中的异步通知与异步 I/O	176
9.1	异步通知的概念与作用	177
9.2	Linux 异步通知编程	177
9.2.1	Linux 信号	177
9.2.2	信号的接收	179
9.2.3	信号的释放	180
9.3	支持异步通知的 globalfifo 驱动	182
9.3.1	在 globalfifo 驱动中增加异步通知	182
9.3.2	在用户空间验证 globalfifo 的异步通知	184
9.4	Linux 2.6 异步 I/O	185
9.4.1	AIO 概念与 GNU C 库函数	185
9.4.2	使用信号作为 AIO 的通知	188
9.4.3	使用回调函数作为 AIO 的通知	189



9.4.4 AIO 与设备驱动	190
9.5 总结	192
第 10 章 中断与时钟	193
10.1 中断与定时器	194
10.2 Linux 中断处理程序架构	195
10.3 Linux 中断编程	196
10.3.1 申请和释放中断	196
10.3.2 使能和屏蔽中断	197
10.3.3 底半部机制	197
10.3.4 实例：S3C6410 实时钟中断	200
10.4 中断共享	202
10.5 内核定时器	203
10.5.1 内核定时器编程	203
10.5.2 内核中延迟的工作 delayed_work	205
10.5.3 实例：秒字符设备	206
10.6 内核延时	210
10.6.1 短延迟	210
10.6.2 长延迟	210
10.6.3 睡着延迟	211
10.7 总结	212
第 11 章 内存与 I/O 访问	213
11.1 CPU 与内存和 I/O	214
11.1.1 内存空间与 I/O 空间	214
11.1.2 内存管理单元 MMU	215
11.2 Linux 内存管理	218
11.3 内存存取	220
11.3.1 用户空间内存动态申请	220
11.3.2 内核空间内存动态申请	221
11.3.3 虚拟地址与物理地址关系	224
11.4 设备 I/O 端口和 I/O 内存的访问	225
11.4.1 Linux I/O 端口和 I/O 内存访问接口	225
11.4.2 申请与释放设备 I/O 端口和 I/O 内存	226

11.4.3 设备 I/O 端口和 I/O 内存访问流程	227
11.4.4 将设备地址映射到用户空间	228
11.5 I/O 内存静态映射	233
11.6 DMA	236
11.6.1 DMA 与 Cache 一致性	236
11.6.2 Linux 下的 DMA 编程	237
11.7 总结	241
第 12 章 工程中的 Linux 设备驱动	242
12.1 platform 设备驱动	243
12.1.1 platform 总线、设备与驱动	243
12.1.2 将 globalfifo 作为 platform 设备	244
12.1.3 platform 设备资源和数据	246
12.2 设备驱动的分层思想	248
12.2.1 设备驱动核心层和例化	248
12.2.2 输入设备驱动	249
12.2.3 RTC 设备驱动	254
12.3 主机驱动与外设驱动分离思想	255
12.3.1 主机、外设驱动分离的意义	255
12.3.2 Linux SPI 主机和设备驱动	256
12.4 设备驱动中的电源管理	260
12.5 misc 设备驱动	262
12.6 基于 sysfs 的设备驱动	263
12.7 Linux 设备驱动的固件加载	265
12.8 Android 设备驱动	266
12.9 总结	269

第 3 篇 Linux 设备驱动实例

第 13 章 Linux 块设备驱动	272
13.1 块设备的 I/O 操作特点	273
13.2 Linux 块设备驱动结构	273
13.2.1 block_device_operations 结构体	273
13.2.2 gendisk 结构体	274



13.2.3	request 与 bio 结构体	276
13.2.4	块设备驱动注册与注销	285
13.3	Linux 块设备驱动的模块加载与卸载	286
13.4	块设备的打开与释放	288
13.5	块设备驱动的 ioctl 函数	288
13.6	块设备驱动的 I/O 请求处理	289
13.6.1	使用请求队列	289
13.6.2	不使用请求队列	291
13.7	实例 1: vmem_disk 驱动	292
13.7.1	vmem_disk 的硬件原理	292
13.7.2	vmem_disk 驱动模块的加载与卸载	293
13.7.3	vmem_disk 设备驱动 block_device_operations 及成员函数	296
13.7.4	vmem_disk I/O 请求处理	298
13.8	实例 2: IDE 硬盘设备驱动	300
13.9	总结	303
第 14 章	Linux 终端设备驱动	304
14.1	终端设备	305
14.2	终端设备驱动结构	307
14.3	终端设备驱动初始化与释放	311
14.3.1	模块加载与卸载函数	311
14.3.2	打开与关闭函数	312
14.4	数据发送和接收	313
14.5	TTY 线路设置	316
14.5.1	线路设置用户空间接口	316
14.5.2	tty 驱动 set_termios 函数	317
14.5.3	tty 驱动的 tiocmget 和 tiocmset 函数	318
14.5.4	tty 驱动 ioctl 函数	319
14.6	UART 设备驱动	320
14.7	printk 和 early_printk console 驱动	325
14.8	实例: S3C6410 串口与 console 驱动	328
14.8.1	S3C6410 串口硬件描述	328
14.8.2	S3C6410 串口 UART 驱动	330
14.8.3	S3C6410 串口 console 驱动	331

14.9 总结	332
第 15 章 Linux 的 I ² C 核心、总线与设备驱动	333
15.1 Linux 的 I ² C 体系结构	334
15.2 Linux I ² C 核心	339
15.3 Linux I ² C 总线驱动	341
15.3.1 I ² C 适配器驱动加载与卸载	341
15.3.2 I ² C 总线通信方法	342
15.4 Linux I ² C 设备驱动	344
15.4.1 Linux I ² C 设备驱动的模块加载与卸载	344
15.4.2 Linux I ² C 设备驱动的数据传输	344
15.4.3 Linux 的 i2c-dev.c 文件分析	345
15.5 S3C6410 I ² C 总线驱动实例	349
15.5.1 S3C6410 I ² C 控制器硬件描述	349
15.5.2 S3C6410 I ² C 总线驱动总体分析	349
15.5.3 S3C6410 I ² C 适配器驱动的模块加载与卸载	350
15.5.4 S3C6410 I ² C 总线通信方法	354
15.6 AT24XX EEPROM 的 I ² C 设备驱动实例	359
15.7 总结	362
第 16 章 Linux 网络设备驱动	363
16.1 Linux 网络设备驱动的结构	364
16.1.1 网络协议接口层	364
16.1.2 网络设备接口层	366
16.1.3 设备驱动功能层	369
16.1.4 网络设备与媒介层	369
16.2 网络设备驱动的注册与注销	369
16.3 网络设备的初始化	371
16.4 网络设备的打开与释放	372
16.5 数据发送流程	373
16.6 数据接收流程	374
16.7 网络连接状态	377
16.8 参数设置和统计数据	378



16.9	DM9000 网卡设备驱动实例	381
16.9.1	DM9000 网卡硬件描述	381
16.9.2	DM9000 网卡驱动设计分析	383
16.10	总结	387
第 17 章	Linux 音频设备驱动	388
17.1	数字音频设备	389
17.2	音频设备硬件接口	390
17.2.1	PCM 接口	390
17.2.2	IIS 接口	390
17.2.3	AC'97 接口	390
17.3	Linux OSS 音频设备驱动	391
17.3.1	OSS 驱动的组成	391
17.3.2	mixer 接口	392
17.3.3	dsp 接口	393
17.3.4	OSS 用户空间编程	394
17.4	Linux ALSA 音频设备驱动	399
17.4.1	ALSA 的组成	399
17.4.2	card 和组件管理	400
17.4.3	PCM 设备	402
17.4.4	控制接口	412
17.4.5	AC97 API 接口	416
17.4.6	ALSA 用户空间编程	418
17.5	Linux ASoC 音频设备驱动	423
17.5.1	ASoC 驱动的组成	423
17.5.2	ASoC Codec 驱动	423
17.5.3	ASoC 平台驱动	426
17.5.4	ASoC 板驱动	429
17.6	S3C6410+WM9714 ASoC 驱动实例	430
17.7	总结	439
第 18 章	LCD 设备驱动	440
18.1	LCD 硬件原理	441
18.2	帧缓冲	443

18.2.1 帧缓冲的概念	443
18.2.2 显示缓冲区与显示点	443
18.2.3 Linux 帧缓冲相关数据结构与函数	444
18.3 Linux 帧缓冲设备驱动结构	450
18.4 帧缓冲设备驱动的模块加载与卸载函数	450
18.5 帧缓冲设备显示缓冲区的申请与释放	452
18.6 帧缓冲设备的参数设置	453
18.6.1 定时参数	453
18.6.2 像素时钟	454
18.6.3 颜色位域	454
18.6.4 固定参数	455
18.7 帧缓冲设备驱动的 fb_ops 成员函数	455
18.8 LCD 设备驱动的读写、mmap 和 ioctl 函数	456
18.9 帧缓冲设备的用户空间访问	461
18.10 Linux 图形用户界面	463
18.10.1 Qt-X11/QtEmbedded/Qtopia	463
18.10.2 Microwindows/Nano-X	467
18.10.3 MiniGUI	469
18.10.4 Android	471
18.11 实例：S3C6410 LCD 设备驱动	474
18.12 总结	478
第 19 章 Flash 设备驱动	479
19.1 Linux Flash 驱动结构	480
19.1.1 Linux MTD 系统层次	480
19.1.2 Linux MTD 系统接口	480
19.1.3 MTD 用户空间编程	485
19.2 NOR Flash 驱动	488
19.3 NAND Flash 驱动	491
19.4 NOR Flash 驱动实例：S3C6410 外围的 NOR Flash 驱动	496
19.5 NAND Flash 驱动实例：S3C6410 外围的 NAND Flash 驱动	497
19.5.1 S3C6410 NAND 控制器硬件描述	497
19.5.2 S3C6410 nand_chip 初始化与 NAND 探测	498
19.6 Flash 文件系统的建立	500



19.6.1	Flash 转换层	500
19.6.2	CramFS	501
19.6.3	JFFS/JFFS2	501
19.6.4	YAFFS/YAFFS2	502
19.6.5	UBI/UBIFS	505
19.7	总结	506
第 20 章	USB 主机与设备驱动	507
20.1	Linux USB 驱动层次	508
20.1.1	主机侧与设备侧 USB 驱动	508
20.1.2	设备、配置、接口、端点	509
20.2	USB 主机控制器驱动	512
20.2.1	USB 主机控制器驱动的整体结构	512
20.2.2	实例：S3C6410 USB 1.1 主机驱动	516
20.3	USB 设备驱动	518
20.3.1	USB 设备驱动整体结构	518
20.3.2	USB 请求块 (URB)	523
20.3.3	探测和断开函数	527
20.3.4	USB 骨架程序	528
20.3.5	实例：USB 键盘驱动	534
20.4	USB UDC 与 gadget 驱动	536
20.4.1	UDC 和 gadget 驱动关键数据结构与 API	536
20.4.2	实例：S3C6410 USB 2.0 的 UDC 驱动	540
20.4.3	实例：file storage gadget 驱动	542
20.5	USB OTG 驱动	544
20.6	总结	545
第 21 章	PCI 设备驱动	547
21.1	PCI 总线与配置空间	548
21.1.1	PCI 总线的 Linux 描述	548
21.1.2	PCI 设备的 Linux 描述	550
21.1.3	PCI 配置空间访问	551
21.1.4	PCI DMA 相关的 API	555
21.1.5	PCI 设备驱动其他常用 API	555

21.2	PCI 设备驱动结构	556
21.2.1	PCI 设备驱动的组成	556
21.2.2	实例：PCI 骨架程序	560
21.3	总结	562

第 4 篇 Linux 设备驱动调试、移植

第 22 章	Linux 设备驱动的调试	564
22.1	Linux 开发环境建设	565
22.1.1	实验室建设	565
22.1.2	工具链	566
22.1.3	串口工具	567
22.2	GDB 调试器用法	570
22.2.1	GDB 基本用法	570
22.2.2	DDD 图形界面调试工具	578
22.3	Linux 内核调试	580
22.4	内核打印信息——printk()	581
22.5	使用/proc	582
22.6	Oops	586
22.7	监视工具	588
22.8	内核调试器	589
22.8.1	kcore	589
22.8.2	KDB	592
22.8.3	KGDB	594
22.9	使用仿真器调试内核	595
22.10	应用程序调试	596
22.11	Linux 性能监控与调优工具	598
22.12	总结	601
第 23 章	Linux 设备驱动的移植	602
23.1	编写可移植的设备驱动	603
23.1.1	可移植的数据类型	603
23.1.2	结构体对界	604
23.1.3	Little Endian 与 Big Endian	605



23.1.4 内存页面大小	605
23.2 巧用同类设备驱动	606
23.2.1 巧用 demo 板驱动	606
23.2.2 巧用类似芯片的驱动程序	606
23.2.3 借用芯片厂商的范例程序	609
23.3 从 Linux 2.4 移植设备驱动到 Linux 2.6	610
23.4 Linux 与其他操作系统之间的驱动移植	618
23.5 Linux 内核的移植	626
23.6 总结	630
参考文献	631

LINUX

第1章 Linux 设备驱动概述及开发环境构建

本章导读

本章将介绍 Linux 设备驱动开发的基本概念，并对本书所基于的平台和开发环境进行讲解。

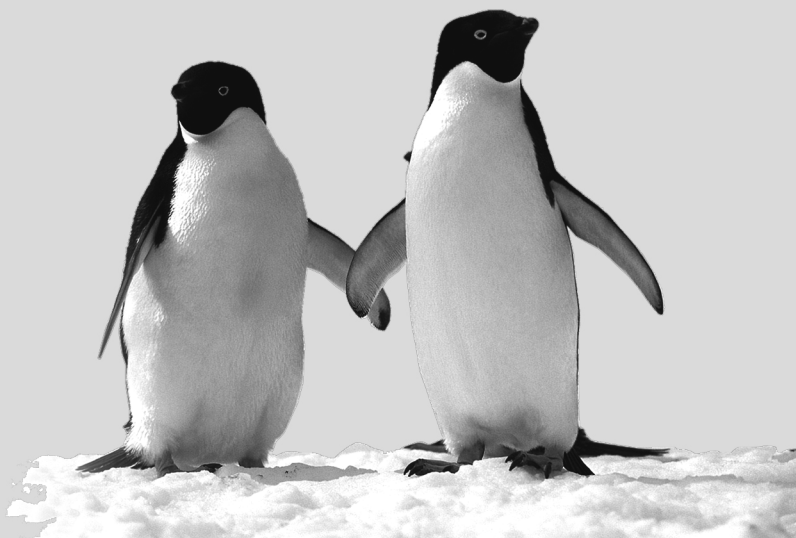
1.1 节阐明了设备驱动的概念和作用。

1.2 节和 1.3 节分别讲解在无操作系统情况下和有操作系统情况下设备驱动的设计，通过对两者不同的分析讲解设备驱动与硬件和操作系统的关系。

1.4 节对 Linux 操作系统的设备驱动进行了概要性的介绍，给出了设备驱动与整个软硬件系统的关系，分析了 Linux 设备驱动的重点、难点和学习方法。

1.5 节对本书所基于的 LDD6410 ARM11 开发板和开发环境的安装进行了介绍。

本章的最后给出了一个设备驱动的“Hello World”实例，即最简单的 LED 驱动在无操作系统情况下和 Linux 操作系统下的实现。



1.1 设备驱动的作用

任何一个计算机系统的运转都是系统中软硬件共同努力的结果，没有硬件的软件是空中楼阁，而没有软件的硬件则只是一堆废铁。硬件是底层基础，是所有软件得以运行的平台，代码最终会落实为硬件上的组合逻辑与时序逻辑。软件则实现了具体应用，它按照各种不同的业务需求而设计，完成了用户的最终诉求。硬件较固定，软件则很灵活，可以适应各种复杂多变的应用。可以说，计算机系统的软硬件互相成就了对方。

但是，软硬件之间同样存在着悖论，那就是软件和硬件不应该互相渗透入对方的领地。为尽可能快速地完成设计，应用软件工程师不想也不必关心硬件，而硬件工程师也难有足够的闲暇和能力来顾及软件。譬如，应用软件工程师在调用套接字发送和接收数据包的时候，不必关心网卡上的中断、寄存器、存储空间、I/O 端口、片选以及其他任何硬件词汇；在使用 `printf()` 函数输出信息的时候，他不用知道底层究竟是怎样把相应的信息输出到屏幕或者串口。

也就是说，应用软件工程师需要看到一个没有硬件的纯粹的软件世界，硬件必须被透明地呈现给他。谁来实现硬件对应用软件工程师的隐形？这个光荣而艰巨的任务就落在了驱动工程师的头上。

对设备驱动最通俗的解释就是“驱使硬件设备行动”。驱动与底层硬件直接打交道，按照硬件设备的具体工作方式，读写设备的寄存器，完成设备的轮询、中断处理、DMA 通信，进行物理内存向虚拟内存的映射等，最终让通信设备能收发数据，让显示设备能显示文字和画面，让存储设备能记录文件和数据。

由此可见，设备驱动充当了硬件和应用软件之间的纽带，它使得应用软件只需要调用系统软件的应用编程接口（API）就可让硬件去完成要求的工作。在系统中没有操作系统的情况下，工程师可以根据硬件设备的特点自行定义接口，如对串口定义 `SerialSend()`、`SerialRecv()`，对 LED 定义 `LightOn()`、`LightOff()`，对 Flash 定义 `FlashWrite()`、`FlashRead()` 等。而在有操作系统的情况下，驱动的架构则由相应的操作系统定义，驱动工程师必须按照相应的架构设计驱动，这样，驱动才能良好地整合入操作系统的内核。

驱动程序沟通着硬件和应用软件，而驱动工程师则沟通着硬件工程师和应用软件工程师。目前，随着通信、电子行业的迅速发展，全世界每天都会有大量的新芯片被生产，大量的新电路板被设计，也因此，会有大量设备驱动需要开发。这些驱动，或运行在简单的单任务环境，或运行在 VxWorks、Linux、Windows 等多任务操作系统环境，发挥着不可替代的作用。

1.2 无操作系统时的设备驱动

并不是任何一个计算机系统都一定要运行操作系统，在许多情况下，操作系统都不必存在。对于功能比较单一、控制并不复杂的系统，譬如 ASIC 内部、公交车的刷卡机、电冰箱、微波炉、简单的手机和小灵通等，并不需要多任务调度、文件系统、内存管理等复杂功能，用单任务架构完全可以良好地支持它们的工作。一个无限循环中夹杂对设备中断的检测或者对设备的轮询是这



种系统中软件的典型架构,如代码清单 1.1。

代码清单 1.1 单任务软件典型架构

```
1 int main(int argc, char* argv[])
2 {
3     while (1)
4     {
5         if (serialInt == 1)
6             /*有串口中断*/
7             {
8                 ProcessSerialInt(); /*处理串口中断*/
9                 serialInt = 0; /*中断标志变量清 0*/
10            }
11        if (keyInt == 1)
12            /*有按键中断*/
13            {
14                ProcessKeyInt(); /*处理按键中断*/
15                keyInt = 0; /*中断标志变量清 0*/
16            }
17        status = CheckXXX();
18        switch (status)
19        {
20            ...
21        }
22        ...
23    }
24 }
```

在这样的系统中,虽然不存在操作系统,但是设备驱动则无论如何都必须存在。一般情况下,每一种设备驱动都会定义为一个软件模块,包含.h文件和.c文件,前者定义该设备驱动的数据结构并声明外部函数,后者进行驱动的具体实现。譬如,可以如代码清单 1.2 那样定义一个串口的驱动。

代码清单 1.2 无操作系统情况下串口的驱动

```
1  /******
2   *serial.h 文件
3   *****/
4  extern void SerialInit(void);
5  extern void SerialSend(const char buf*,int count);
6  extern void SerialRecv(char buf*,int count);
7
8  /******
9   *serial.c 文件
10  *****/
11  /*初始化串口*/
12  void SerialInit(void)
13  {
14      ...
15  }
16  /*串口发送*/
17  void SerialSend(const char buf*,int count)
18  {
19      ...
20  }
21  /*串口接收*/
```

```

22 void SerialRecv(char buf*,int count)
23 {
24     ...
25 }
26 /*串口中断处理函数*/
27 void SerialIsr(void)
28 {
29     ...
30     serialInt = 1;
31 }

```

其他模块想要使用这个设备的时候，只需要包含设备驱动的头文件 `serial.h`，然后调用其中的外部接口函数。如我们要从串口上发送“Hello World”字符串，使用语句 `SerialSend (“Hello World”, 11)` 即可。

由此可见，在没有操作系统的情况下，设备驱动的接口被直接提交给了应用软件的工程师，应用软件没有跨越任何层次就直接访问了设备驱动的接口。驱动包含的接口函数也与硬件的功能直接吻合，没有任何附加功能。图 1.1 所示为无操作系统情况下硬件、驱动与应用软件的关系。

有的工程师把单任务系统设计成了如图 1.2 所示的结构，即设备驱动和具体的应用软件模块之间平等，驱动中包含了业务层面上的处理，这显然是不合理的，不符合软件设计中高内聚、低耦合的要求。

另一种不合理的设计是直接在应用中操作硬件的寄存器，而不单独设计驱动模块，如图 1.3 所示。这种设计意味着系统中不存在或未能充分利用可被重用的驱动代码。

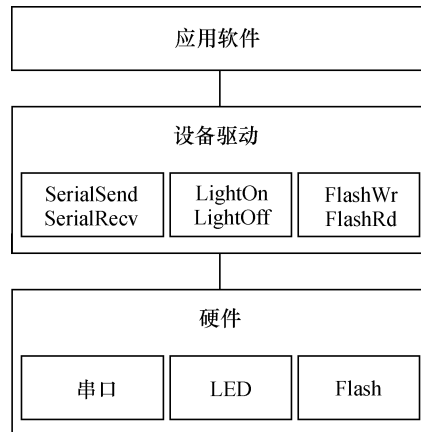


图 1.1 无操作系统时硬件、驱动和应用软件的关系

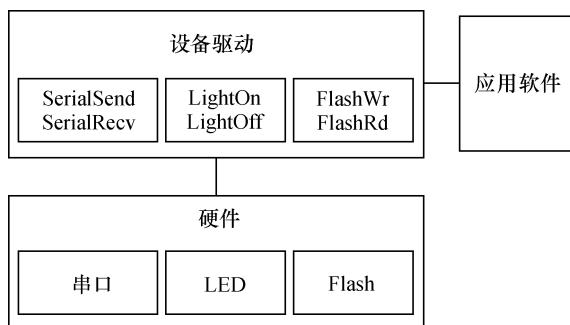


图 1.2 驱动与应用高耦合的不合理设计

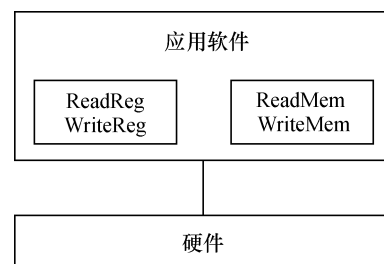


图 1.3 应用直接访问硬件的不合理设计

1.3 有操作系统时的设备驱动

1.2 节中我们看到一个干净利落的设备驱动，它直接运行在硬件之上，不与任何操作系统关联。



当系统中包含操作系统后, 设备驱动会变得怎样?

首先, 无操作系统时设备驱动的硬件操作工作仍然是必不可少的, 没有这一部分, 驱动不可能与硬件打交道。

其次, 我们还需要将驱动融入内核。为了实现这种融合, 必须在所有设备的驱动中设计面向操作系统内核的接口, 这样的接口由操作系统规定, 对一类设备而言结构一致, 独立于具体的设备。

由此可见, 当系统中存在操作系统的时候, 驱动变成了连接硬件和内核的桥梁。如图 1.4, 操作系统的存在势必要求设备驱动附加更多的代码和功能, 把单一的“驱使硬件设备行动”变成了操作系统内与硬件交互的模块, 它对外呈现为操作系统的 API, 不再给应用软件工程师直接提供接口。

那么我们要问, 有了操作系统之后, 驱动反而变得复杂, 那要操作系统干什么?

首先, 一个复杂的软件系统需要处理多个并发的任务, 没有操作系统, 想完成多任务并发是很困难的。

其次, 操作系统给我们提供内存管理机制。一个典型的例子是, 对于多数含 MMU 的处理器而言, Windows、Linux 等操作系统可以让每个进程都可以独立地访问 4GB 的内存空间。

上述优点似乎并没有体现在设备驱动身上, 操作系统的存在给设备驱动究竟带来了什么实质的好处?

简而言之, 操作系统通过给驱动制造麻烦来达到给上层应用提供便利的目的。当驱动都按照操作系统给出的独立于设备的接口而设计, 那么, 应用程序将可使用统一的系统调用接口来访问各种设备。对于类 UNIX 的 VxWorks、Linux 等操作系统而言, 当应用程序通过 write()、read() 等函数读写文件就可访问各种字符设备和块设备, 而不论设备的具体类型和工作方式, 那将是怎样的便利?

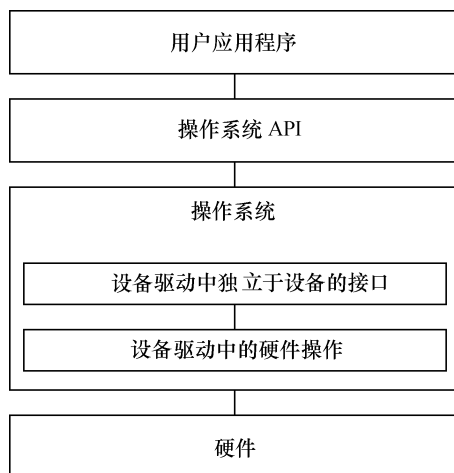


图 1.4 硬件、驱动、操作系统和应用程序的关系

1.4 Linux 设备驱动

1.4.1 设备的分类及特点

计算机系统的硬件主要由 CPU、存储器和外设组成。随着 IC 制作工艺的发展, 目前, 芯片的集成度越来越高, 往往在 CPU 内部就集成了存储器和外设适配器。譬如, 相当多的 ARM、PowerPC、MIPS 等处理器都集成了 UART、I²C 控制器、USB 控制器、SDRAM 控制器等, 有的处理器还集成了片内 RAM 和 Flash。

驱动针对的对象是存储器和外设 (包括 CPU 内部集成的存储器和外设), 而不是针对 CPU 核。Linux 将存储器和外设分为 3 个基础大类。

- 字符设备。
- 块设备。

- 网络设备。

字符设备指那些必须以串行顺序依次进行访问的设备，如触摸屏、磁带驱动器、鼠标等。块设备可以用任意顺序进行访问，以块为单位进行操作，如硬盘、软驱等。字符设备不经过系统的快速缓冲，而块设备经过系统的快速缓冲。但是，字符设备和块设备并没有明显的界限，如对于 Flash 设备，符合块设备的特点，但是我们仍然可以把它作为一个字符设备来访问。

字符设备和块设备的驱动设计呈现出很大的差异，但是对于用户而言，他们都使用文件系统的操作接口 `open()`、`close()`、`read()`、`write()` 等进行访问。

在 Linux 系统中，网络设备面向数据包的接收和发送而设计，它并不对应于文件系统的节点。内核与网络设备的通信与内核和字符设备、网络设备的通信方式完全不同。

另外一种设备分类方法中所称的 I²C 驱动、USB 驱动、PCI 驱动、LCD 驱动等本身可归纳入 3 个基础大类，但是对于这些复杂的设备，Linux 也定义了独特的驱动体系结构。

1.4.2 Linux 设备驱动与整个软硬件系统的关系

如图 1.5 所示，除网络设备外，字符设备与块设备都被映射到 Linux 文件系统的文件和目录，通过文件系统的系统调用接口 `open()`、`write()`、`read()`、`close()` 等即可访问字符设备和块设备。所有的字符设备和块设备都被统一地呈现给用户。块设备比字符设备复杂，在它上面会首先建立一个磁盘/Flash 文件系统，如 FAT、EXT3、YAFFS2、JFFS2、UBIFS 等。FAT、EXT3、YAFFS2、JFFS2、UBIFS 定义了文件和目录在存储介质上的组织。

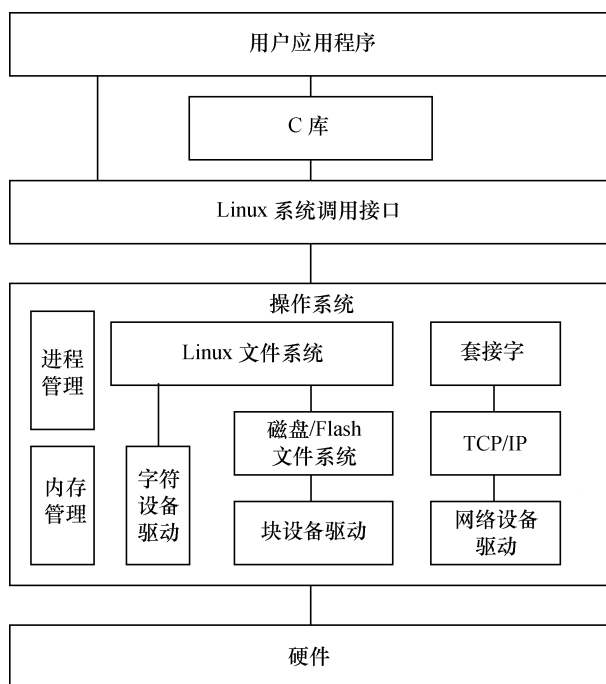


图 1.5 Linux 设备驱动与整个软硬件系统的关系

应用程序可以使用 Linux 的系统调用接口编程，但也可使用 C 库函数，出于代码可移植性的目的，后者更值得推荐。C 库函数本身也通过系统调用接口而实现，如 C 库函数 `fopen()`、`fwrite()`、



fread()、fclose()分别会调用操作系统的 API open()、write()、read()、close()。

1.4.3 Linux 设备驱动的重点、难点

Linux 设备驱动的学习是一项浩繁的工程, 包含如下的重点、难点。

- 编写 Linux 设备驱动要求工程师有非常好的硬件基础, 懂得 SRAM、Flash、SDRAM、磁盘的读写方式, UART、I²C、USB 等设备的接口以及轮询、中断、DMA 的原理, PCI 总线的工作方式以及 CPU 的内存管理单元 (MMU) 等。
- 编写 Linux 设备驱动要求工程师有非常好的 C 语言基础, 能灵活地运用 C 语言的结构体、指针、函数指针及内存动态申请和释放等。
- 编写 Linux 设备驱动要求工程师有一定的 Linux 内核基础, 虽然并不要求工程师对内核各个部分有深入的研究, 但至少明白驱动与内核的接口。尤其是对于块设备、网络设备、Flash 设备、串口设备等复杂设备, 内核定义的驱动体系架构本身就非常复杂。
- 编写 Linux 设备驱动要求工程师有非常好的多任务并发控制和同步的基础, 因为在驱动中会大量使用自旋锁、互斥、信号量、等待队列等并发与同步机制。

上述经验值的获取并非朝夕之事, 因此要求我们有足够的学习恒心和毅力。对这些重点、难点, 本书都会有相应章节进行讲解。

动手实践永远是学习任何软件开发的最好方法, 学习 Linux 设备驱动也不例外。因此, 本书专门配备了一款基于 S3C6410 的 ARM11 开发板 LDD6410 (全称 Linux Device Drivers 6410, 即 Linux 设备驱动开发 6410 专用板), 本书中的所有实例均可在该电路板上直接执行。

阅读经典书籍和参与 Linux 社区的讨论也是非常好的学习方法。Linux 内核源代码中包含了一个 Documentation 目录, 其中包含了一批内核设计的文档, 全部是文本文件。很遗憾, 这些文档的组织不太好, 内容也不够细致。本书的参考目录中给出了一些优秀的参考书籍和 Linux 网站, 并进行了简单的介绍。

学习 Linux 设备驱动的一个注意事项是要避免管中窥豹、只见树木不见森林, 因为各类 Linux 设备驱动都从属于一个 Linux 设备驱动的架构, 单纯而片面地学习几个函数、几个数据结构是不可能理清驱动中各组成部分之间的关系的。因此, Linux 驱动的分析方法是点面结合, 将对函数和数据结构的理解放在整体架构的背景之中。这是本书各章节讲解驱动的方法。

1.5 Linux 设备驱动开发环境构建

1.5.1 PC 上的 Linux 环境

本书配套光盘提供了一个 Ubuntu 的 VirtualBox 虚拟机映像, 该虚拟机上安装了所有本书涉及的源代码、工具链和各种开发工具, 读者无需再安装和配置任何环境。该虚拟机可运行于 Windows 等操作系统中, 运行方法如下。

(1) 解压缩安装盘内的虚拟机磁盘映像 virtual-disk.rar 到本地硬盘得到 virtual-disk.vdi (至少需要 16GB 的空闲磁盘空间)。

- (2) 安装安装盘内的 VirtualBox 虚拟机软件。
- (3) 建立一个虚拟机。
- ① 单击“新建”按钮，指定虚拟机使用 Linux Ubuntu 系统，如图 1.6 所示。

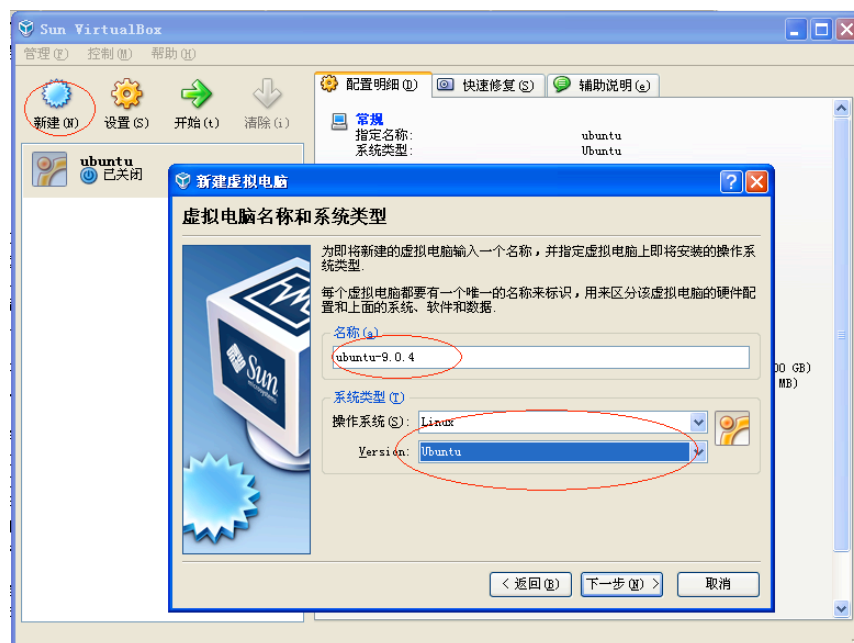


图 1.6 VirtualBox 指定使用 Ubuntu

- ② 单击“下一步”按钮，如图 1.7 所示，使用推荐的内存 384MB。



图 1.7 VirtualBox 中内存设定



③ 指定虚拟机磁盘映像为第一步解压缩得到的 virtual-disk.vdi，如图 1.8 所示。

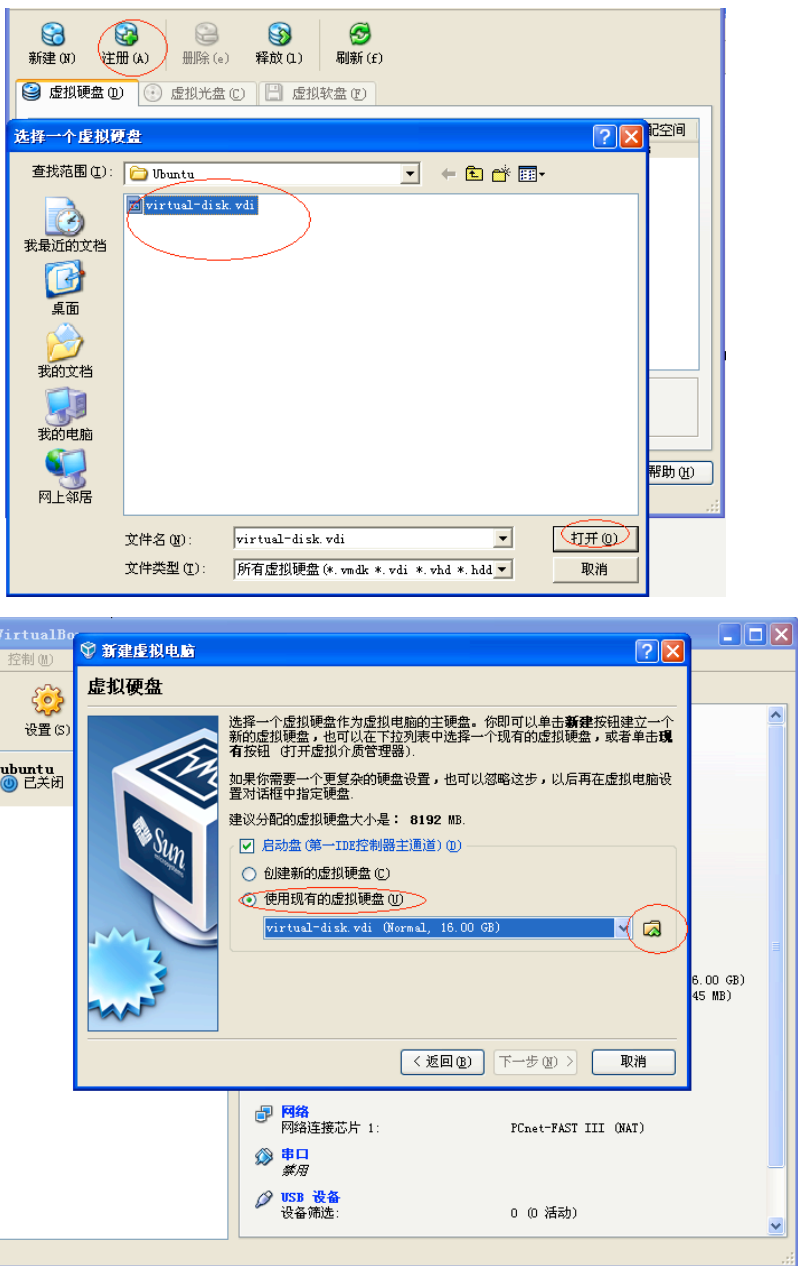


图 1.8 VirtualBox 中磁盘设定

④ 完成设置，如图 1.9 所示。

之后就可以启动虚拟机，账号和密码都是“li hacker”。本书配套源代码都位于 li hacker 主目录的 develop 目录下，几个主要项目针对/home/li hacker/develop/的子目录如下。

LDD6410 开发板内核源代码：svn/ldd6410-2-6-28-read-only/linux-2.6.28-samsung。

LDD6410 开发板 U-BOOT 源代码：svn/ldd6410-read-only/s3c-u-boot-1.1.6。

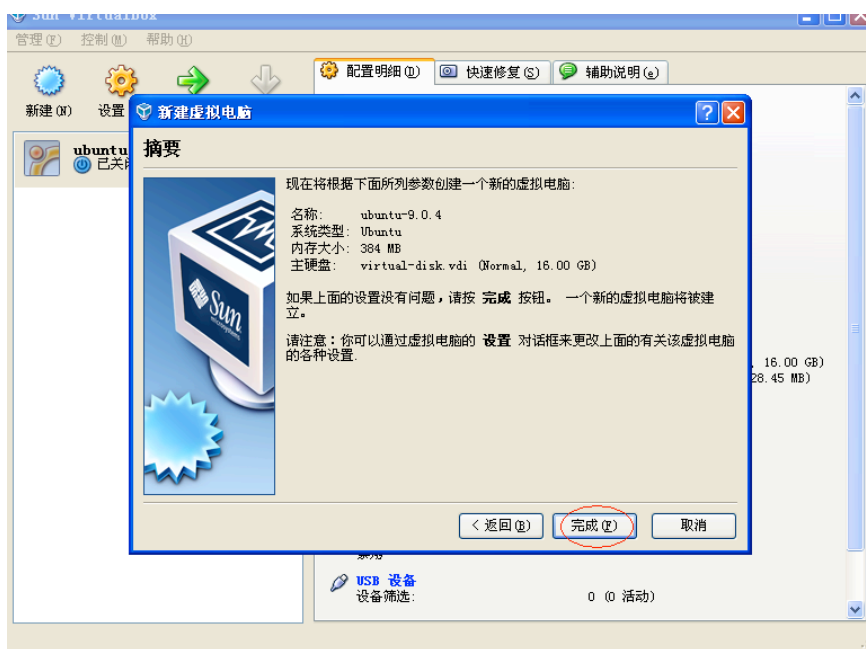


图 1.9 VirtualBox 中完成设定

LDD6410 开发板文件系统用的 busybox、jpegview、mplayer、appweb 等：svn/ldd6410-read-only/utls。

LDD6410 开发板及常用 Linux 用户空间驱动测试程序：svn/ldd6410-read-only/tests。

书中 globalmem、globalfifo 等驱动实例：svn/ldd6410-read-only/training/kernel。

Android 的源代码：git/myandroid。

NDK：android-ndk-r3。

eclipse：单击桌面上的“android-eclipse”图标，即可运行附带 ADT 的 eclipse 开发工具。

1.5.2 LDD6410 开发板

LDD6410 是本书专配的一款高端 ARM11 处理器开发板（其结构如图 1.10 所示，实物如图 1.11 所示），采用三星公司最新推出 S3C6410 处理器，芯片拥有强大的内部资源和视频处理能力，板上集成了丰富的外围接口，其主要特点如下。

- (1) 运行于 533MHz 的 ARM11 处理器（最高主频可达到 667MHz）。
- (2) 运行于 266MHz 的 DDR 内存，128MB。
- (3) 1MB NOR Flash。
- (4) 256MB NAND Flash。
- (5) WM9714 AC97 声卡。
- (6) VGA 输出接口（可达 1024×768@60Hz）。
- (7) TV 输出接口。
- (8) USB 2.0 OTG 接口及 USB 1.1 host 接口。

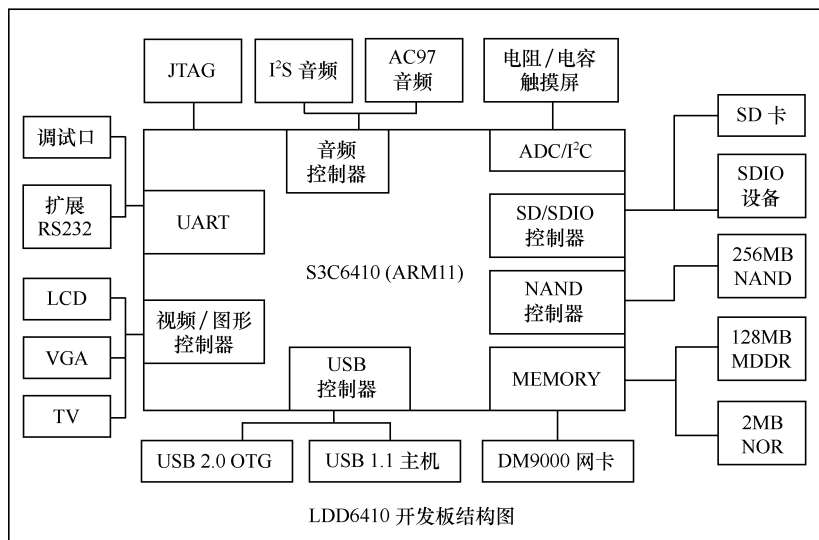


图 1.10 LDD6410 的结构图

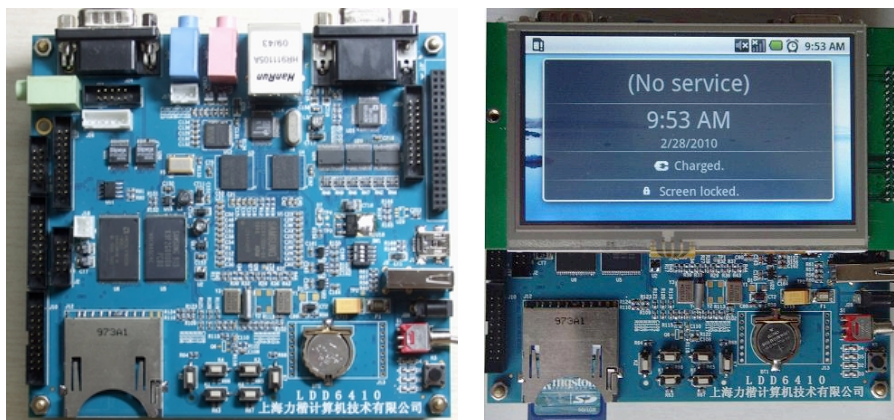


图 1.11 LDD6410 实物图

- (9) SD/SDIO 接口，支持 SD 卡和 SDIO 设备。
- (10) DM9000 百兆网卡。
- (11) 4.3 寸 LCD（分辨率为 480×272）、触摸屏。
- (12) S3C6410 芯片内嵌图形加速，JPEG、多媒体编解码。
- (13) 6 个 GPIO 按键。
- (14) 可扩展 Camera、WiFi、3G modem 等模块。
- (15) 可扩展外部矩阵键盘。

配套电路板提供了如下软件。

- (1) 工具链：提供了 arm-linux-gcc、arm-linux-gdb、gdbserver、strace 用于 Android 开发的 eclipse（带 ADT 插件）、JDK 和 NDK。
- (2) U-BOOT：U-BOOT 源代码包含独立的 LDD6410 文件，支持从 SD 卡、NAND 启动，支持 DM9000 网卡引导。

(3) Linux 内核、BSP 和驱动: Linux 2.6.28 内核、源代码, 包含独立的 LDD6410 BSP 和完整的设备驱动。

(4) 文件系统: 基于新版 Busybox 1.15.1, 文件系统集成 jpegview、mplayer、appweb 等大量应用, 集成了按键、鼠标、触摸屏、LCD 等测试程序, 作为驱动的用户应用案例。

(5) Android: 提供 Android 源代码和文件系统、内核电源管理补丁源代码、内核 Android 驱动源代码。LDD6410 的 Android 系统支持按键、触摸屏和鼠标操作, 支持使用 LCD 和 VGA 进行显示。

(6) QT: LDD6410 支持 Qt/Embedded 4.5.3, 移植了 Ts_lib 和 Tslib, ts_calibration, 支持使用触摸屏进行操作。

LDD6410 支持从 SD 卡或 NAND 启动, 通过电路板上的 SW1 可设置 LDD6410 的启动模式。从 SD 卡启动设备为全 ON; 从 NAND 启动时, 将 1、2 设置为 ON, 3、4 设置为 OFF。

LDD6410 开发板的详细使用方法, 请见配套光盘中的“LDD6410 开发板用户手册”。

1.5.3 工具链安装

本书配套光盘的虚拟机映像中已经安装好了 LDD6410 的工具链, 读者如果想在其他环境中安装, 只需要从 <http://ldd6410.googlecode.com/files/cross-4.2.2-eabi.tar.bz2> 下载。LDD6410 开发板工具链为 S3C6410X-ToolChain4.2.2-EABI-V0.0-cross-4.2.2-eabi.tar。安装步骤如下。

(1) 解压上述工具链获得文件夹: 4.2.2-eabi/。

(2) 在/usr/local/下面创建目录 arm/ (注意, 最好是放到这个目录, 不然在以后的编译过程中可能出现一些错误)。

(3) 将目录 4.2.2-eabi/移动到/usr/local/arm/下面。

(4) 设置环境变量。

编辑/etc/profile 文件, 在文件末尾添加:

```
PATH="$PATH:/usr/local/arm/4.2.2-eabi/usr/bin"
export PATH
```

使环境变量生效, 在终端输入命令:

```
source /etc/profile
```

另外, 也可以通过修改 home 目录的.bashrc 来将/usr/local/arm/4.2.2-eabi/usr/bin 添加到 PATH:

```
export PATH=/usr/local/arm/4.2.2-eabi/usr/bin:$PATH
```

(5) 测试环境变量是否设置成功。

在终端输入: echo \$PATH, 如果输出的路径中包含了/usr/local/arm/4.2.2-eabi/usr/bin, 则说明环境变量设置成功。

(6) 测试交叉编译工具链。

在终端输入“arm-linux-gcc -v”, 显示如下:

```
Using built-in specs.
Target: arm-unknown-linux-gnueabi
Configured with:
/home/scsuh/workplace/coffee/buildroot-20071011/toolchain_build_arm
/gcc-4.2.2/configure --prefix=/usr --build=i386-pc-linux-gnu --host=i386-pc-linux-gnu
--target=arm-unknown-linux-gnueabi --enable-languages=c,c++ --with-sysroot=/usr/local
/arm/4.2.2-eabi/ --with-build-time-tools=/usr/local/arm/4.2.2-eabi/usr/arm-unknown-linux-
gnueabi/bin --disable-cxa_atexit --enable-target-optspace --with-gnu-ld --enable-shared
```



```
--with-gmp=/usr/local/arm/4.2.2-eabi/gmp --with-mpfr=/usr/local/arm/4.2.2-eabi/mpfr
--disable-nls --enable-threads --disable-multilib --disable-largefile --with-arch=armv4t
--with-float=soft --enable-cxx-flags=-msoft-float
Thread model: posix gcc version 4.2.2
```

说明交叉编译工具链已经安装成功。

ldd6410-debug-tools.tar.gz 调试工具包包含了 strace、gdbserver 和 arm-linux-gdb, 其中 strace、gdbserver 用于目标板文件系统, arm-linux-gdb 运行于主机端, 对目标板上的内核、内核模块应用程序进行调试。

下载地址为 <http://ldd6410.googlecode.com/files/ldd6410-debug-tools.tar.gz>, 光盘目录为 toolchains/ldd6410-debug-tools.tar.gz。

解压 ldd6410-debug-tools.tar.gz, 将其中的 arm-linux-gdb 放入主机上 arm-linux-gcc 所在的目录 /usr/local/arm/4.2.2-eabi/usr/bin/。

而 strace、gdbserver 则可根据需要放入目标机根文件系统的 /usr/sbin 目录。

1.5.4 主机端 nfs 和 tftp 服务安装

本书配套光盘的虚拟机映像中已经安装好了 nfs 和 tftp, LDD6410 可使用 tftp 或 nfs 文件系统与主机通过网口交互。如果用户想在其他环境下自行安装, 对于 Ubuntu 或 Debian 用户而言, 在主机端可通过如下方法安装 tftp 服务:

```
sudo apt-get install tftpd-hpa
```

开启 tftp 服务:

```
sudo /etc/init.d/tftpdhpa start
Starting HPA's tftpd: in.tftpd.
```

对于 Ubuntu 或 Debian 用户而言, 在主机端可通过如下方法安装 nfs 服务:

```
apt-get install nfs-kernel-server
sudo mkdir /home/nfs
sudo chmod 777 /home/nfs
```

运行 “sudo vim /etc/exports” 或 “sudo gedit /etc/exports”, 修改该文件内容为:

```
/home/nfs *(sync,rw)
```

运行 exportfs rv 开启 NFS 服务:

```
/etc/init.d/nfs-kernel-server restart
```

1.5.5 源代码阅读和编辑

源代码是学习 Linux 的最权威资料, 在 Windows 上阅读 Linux 源代码的最佳工具是 Source Insight, 在其中建立一个工程, 并将 Linux 的所有源代码加入该工程, 同步这个工程之后, 我们将可以非常方便地在代码之间进行关联阅读, 如图 1.12 所示。

网站 <http://lxr.linux.no/> 提供了内核版本 2.6.11 到最新版 Linux 源代码的交叉索引, 在其中输入 Linux 内核中的函数、数据结构或变量的名称就可以直接得到以超链接形式给出的定义和引用它的所有位置。还有一些网站也提供了 Linux 内核中函数、变量和数据结构的搜索能力, 在 google 中搜索 “linux identifier search” 可得。

在 Linux 主机上阅读和编辑 Linux 源码的常用方式是 vim + cscope 或者 vim + ctags, vim 是一个文本编辑器, 而 cscope 和 ctags 则可建立代码索引, 建议读者尽快使用基于文本界面全键盘操作的 vim 编辑器, 如图 1.13 所示。

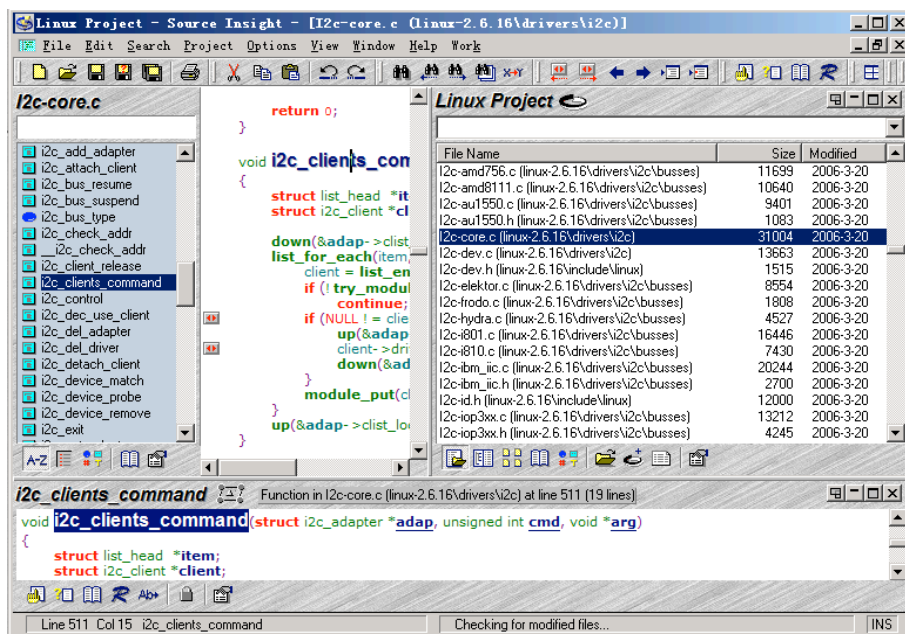


图 1.12 在 Source Insight 中阅读 Linux 源代码

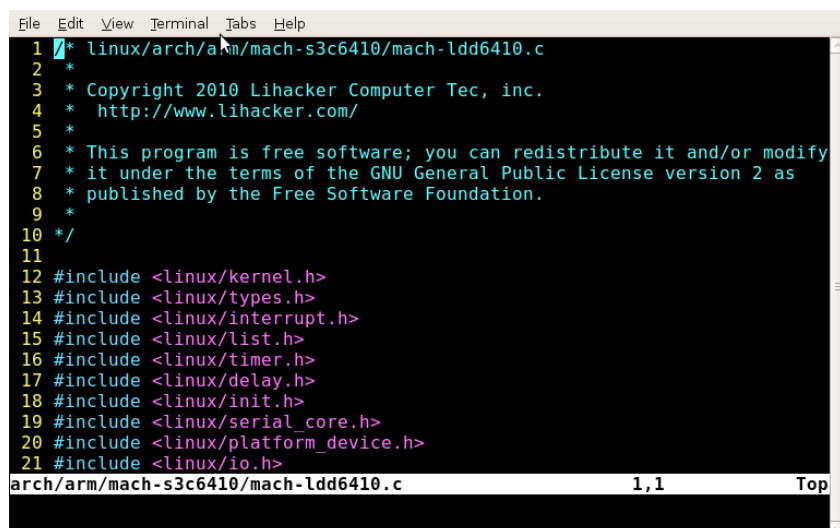


图 1.13 vim 编辑器

1.6 设备驱动 Hello World : LED 驱动

1.6.1 无操作系统时的 LED 驱动

在嵌入式系统的设计中，LED 一般直接由 CPU 的 GPIO（通用可编程 I/O 口）控制。GPIO



一般由两组寄存器控制,即一组控制寄存器和一组数据寄存器。控制寄存器可设置 GPIO 口的工作方式为输入或是输出。当引脚被设置为输出时,向数据寄存器的对应位写入 1 和 0 会分别在引脚上产生高电平和低电平;当引脚设置为输入时,读取数据寄存器的对应位可获得引脚上的电平为高或低。

在本例子中,我们屏蔽具体 CPU 的差异,假设在 GPIO_REG_CTRL 物理地址处的控制寄存器处的第 n 位写入 1 可设置 GPIO 为输出,在地址 GPIO_REG_DATA 物理地址处的数据寄存器的第 n 位写入 1 或 0 可在引脚上产生高或低电平,则无操作系统的情况下,设备驱动为代码清单 1.3。

代码清单 1.3 无操作系统时的 LED 驱动

```
1 #define reg_gpio_ctrl *(volatile int *) (ToVirtual(GPIO_REG_CTRL))
2 #define reg_gpio_data *(volatile int *) (ToVirtual(GPIO_REG_DATA))
3 /*初始化 LED*/
4 void LightInit(void)
5 {
6     reg_gpio_ctrl |= (1 << n); /*设置 GPIO 为输出*/
7 }
8
9 /*点亮 LED*/
10 void LightOn(void)
11 {
12     reg_gpio_data |= (1 << n); /*在 GPIO 上输出高电平*/
13 }
14
15 /*熄灭 LED*/
16 void LightOff(void)
17 {
18     reg_gpio_data &= ~(1 << n); /*在 GPIO 上输出低电平*/
19 }
```

上述程序中的 LightInit()、LightOn()、LightOff()都直接作为驱动提供给应用程序的外部接口函数。程序中 ToVirtual()的作用是当系统启动了硬件 MMU 之后,根据物理地址和虚拟地址的映射关系,将寄存器的物理地址转化为虚拟地址。

1.6.2 Linux 下的 LED 驱动

在 Linux 下,可以使用字符设备驱动的框架来编写对应于代码清单 1.3 的 LED 设备驱动(这里仅仅是为了讲解的方便,到后文我们会发现,内核中实际实现了一个提供 sysfs 结点的 GPIO LED 驱动,位于 drivers/leds/leds-gpio.c),操作硬件的 LightInit()、LightOn()、LightOff()函数仍然需要,但是,遵循 Linux 编程的命名习惯,重新将其命名为 light_init()、light_on()、light_off()。这些函数将被 LED 设备驱动中独立于设备的针对内核的接口进行调用,代码清单 1.4 给出了 Linux 下 LED 的驱动,此时读者并不需要能读懂这些代码。

代码清单 1.4 Linux 操作系统下 LED 的驱动

```
1 #include .../*包含内核中的多个头文件*/
2
3 /*设备结构体*/
4 struct light_dev {
5     struct cdev cdev; /*字符设备 cdev 结构体*/
6 }
```

```

5     unsigned char vaule; /*LED 亮时为 1, 熄灭时为 0, 用户可读写此值*/
6 };

7 struct light_dev *light_devp;
8 int light_major = LIGHT_MAJOR;

9 MODULE_AUTHOR("Barry Song <21cnbao@gmail.com>");
10 MODULE_LICENSE("Dual BSD/GPL");
11 /*打开和关闭函数*/
12 int light_open(struct inode *inode, struct file *filp)
13 {
14     struct light_dev *dev;
15     /* 获得设备结构体指针 */
16     dev = container_of(inode->i_cdev, struct light_dev, cdev);
17     /* 让设备结构体作为设备的私有信息 */
18     filp->private_data = dev;
19     return 0;
20 }

21 int light_release(struct inode *inode, struct file *filp)
22 {
23     return 0;
24 }

25 /*读写设备:可以不需要 */
26 ssize_t light_read(struct file *filp, char __user *buf, size_t count,
27     loff_t *f_pos)
28 {
29     struct light_dev *dev = filp->private_data; /*获得设备结构体 */
30     if (copy_to_user(buf, &(dev->value), 1))
31         return -EFAULT;

32     return 1;
33 }

34 ssize_t light_write(struct file *filp, const char __user *buf, size_t count,
35     loff_t *f_pos)
36 {
37     struct light_dev *dev = filp->private_data;

38     if (copy_from_user(&(dev->value), buf, 1))
39         return -EFAULT;

40     /*根据写入的值点亮和熄灭 LED*/
41     if (dev->value == 1)
42         light_on();
43     else
44         light_off();

45     return 1;
46 }

47 /* ioctl 函数 */
48 int light_ioctl(struct inode *inode, struct file *filp, unsigned int cmd,
49     unsigned long arg)

```



```
50 {
51     struct light_dev *dev = filp->private_data;

52     switch (cmd) {
53     case LIGHT_ON:
54         dev->value = 1;
55         light_on();
56         break;
57     case LIGHT_OFF:
58         dev->value = 0;
59         light_off();
60         break;
61     default:
62         /* 不能支持的命令 */
63         return -ENOTTY;
64     }

65     return 0;
66 }

67 struct file_operations light_fops = {
68     .owner = THIS_MODULE,
69     .read = light_read,
70     .write = light_write,
71     .ioctl = light_ioctl,
72     .open = light_open,
73     .release = light_release,
74 };

75 /*设置字符设备 cdev 结构体*/
76 static void light_setup_cdev(struct light_dev *dev, int index)
77 {
78     int err, devno = MKDEV(light_major, index);
79     cdev_init(&dev->cdev, &light_fops);
80     dev->cdev.owner = THIS_MODULE;
81     dev->cdev.ops = &light_fops;
82     err = cdev_add(&dev->cdev, devno, 1);
83     if (err)
84         printk(KERN_NOTICE "Error %d adding LED%d", err, index);
85 }

86 /*模块加载函数*/
87 int light_init(void)
88 {
89     int result;
90     dev_t dev = MKDEV(light_major, 0);
91     /* 申请字符设备号*/
92     if (light_major)
93         result = register_chrdev_region(dev, 1, "LED");
94     else {
95         result = alloc_chrdev_region(&dev, 0, 1, "LED");
96         light_major = MAJOR(dev);
97     }
98     if (result < 0)
99         return result;
```



```

100     /* 分配设备结构体的内存 */
101     light_devp = kmalloc(sizeof(struct light_dev), GFP_KERNEL);
102     if (!light_devp) {
103         result = -ENOMEM;
104         goto fail_malloc;
105     }
106     memset(light_devp, 0, sizeof(struct light_dev));
107     light_setup_cdev(light_devp, 0);
108     light_gpio_init();
109     return 0;

110 fail_malloc:
111     unregister_chrdev_region(dev, light_devp);
112     return result;
113 }

114 /*模块卸载函数*/
115 void light_cleanup(void)
116 {
117     cdev_del(&light_devp->cdev); /*删除字符设备结构体*/
118     kfree(light_devp); /*释放在 light_init 中分配的内存*/
119     unregister_chrdev_region(MKDEV(light_major, 0), 1); /*删除字符设备*/
120 }

121 module_init(light_init);
122 module_exit(light_cleanup);

```

上述代码的行数与代码清单 1.3 已经不能比拟,除了代码清单 1.3 中的硬件操作函数仍然需要外,代码清单 1.4 中还包含了大量对我们暂时陌生的元素,如结构体 `file_operations`、`cdev`, Linux 内核模块声明用的 `MODULE_AUTHOR`、`MODULE_LICENSE`、`module_init`、`module_exit`, 以及用于字符设备注册、分配和注销用的函数 `register_chrdev_region()`、`alloc_chrdev_region()`、`unregister_chrdev_region()` 等。我们也不能理解为什么驱动中要包含 `light_init()`、`light_cleanup()`、`light_read()`、`light_write()` 等函数。

此时,我们只需要有一个感性认识,那就是,上述暂时陌生的元素都是 Linux 内核给字符设备定义的为实现驱动与内核接口而定义的。Linux 对各类设备的驱动都定义了类似的数据结构和函数。

1.7 全书结构

本书第 1 篇给您打下 Linux 设备驱动的基础。第 1 章简要地介绍了设备驱动的作用,并从无操作系统的设备驱动引出了 Linux 操作系统下的设备驱动,介绍了本书所基于的开发环境。第 2 章系统地讲解了一个 Linux 驱动工程师应该掌握的硬件知识,为工程师打下 Linux 驱动编程的硬件基础,讲解了各种类型的 CPU、存储器和常见的外设,并阐述了硬件时序分析方法和数据手册阅读方法。第 3 章将 Linux 设备驱动放在 Linux 2.6 内核背景中进行讲解,说明 Linux 内核的编程方法。由于驱动编程也在内核编程的范畴,因此,这一章实质是为编写 Linux 设备驱动打下软件基础。



第 2 篇讲解 Linux 设备驱动编程的基础理论、字符设备驱动及设备驱动设计中涉及的并发控制、同步等问题。第 4、5 章分别讲解 Linux 内核模块和 Linux 设备文件系统，第 6~9 章以虚拟设备 `globalmem` 和 `globalfifo` 为主线，逐步给其添加高级控制功能，第 10、11 章分别阐述 Linux 驱动编程中所涉及的中断和定时器、内核和 I/O 操作处理方法，本篇的第 12 章讲解了 Linux 设备驱动工程化的一些问题，属于承前启后的一章。

第 3 篇剖析复杂设备驱动的体系架构，每一章都给出了具体的实例。所涉及的设备包括块设备、终端设备、I²C 适配器与 I²C 设备、网络设备、PCI 设备、USB 设备、LCD 设备、Flash 设备等。这一部分的讲解方法是抽象与具体相结合，先以模板的形式给出各种设备驱动的设计，然后用具体实例设备的驱动填充对应的模板。

第 4 篇分析了 Linux 设备驱动的调试和移植方法。由于在 Linux 设备驱动的设计工作中人们强调多快好省，因此，如果能方便地把现有的其他平台中的驱动移植到 Linux 2.6 平台，或者将类似设备的驱动进行简单修改就运用于新的设备，那将会极大地缩短工程的实施时间。本书的最后几章对 Linux 设备驱动移植中涉及的理论以及移植的技巧进行了讲解。

LINUX

第2章

驱动设计的硬件基础

本章导读

本章讲述一个底层驱动工程师必备的硬件基础，给出了嵌入式系统硬件原理及分析方法的一个完整而简洁的全景视图。

2.1 节描述了微控制器、微处理器、数字信号处理器以及应用于特定领域的处理器各自的特点，分析了处理器的体系架构和指令集。

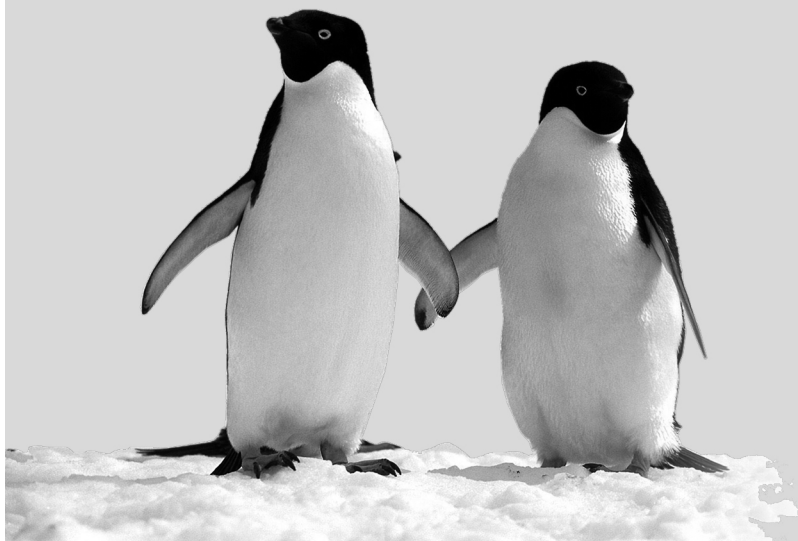
2.2 节对嵌入式系统中所使用的各类存储器与 CPU 的接口、应用领域及特点进行了归纳整理。

2.3 节分析了常见的外设接口与总线的工作方式，包括串口、I²C、USB、以太网接口、ISA、PCI 和 cPCI 等。

嵌入式系统硬件电路中经常会使用 CPLD 和 FPGA，作为驱动工程师，我们不需要掌握 CPLD 和 FPGA 的开发方法，但是需要知道它们在电路中能完成什么工作，2.4 节讲解了这项内容。

2.5~2.7 节给出了在实际项目开发过程中硬件分析的方法，包括如何进行原理图分析、时序分析及如何快速地从芯片手册获取有效信息。

2.8 节讲解了调试过程中常用仪器仪表的使用方法，涉及万用表、示波器和逻辑分析仪。





2.1 处理器

2.1.1 通用处理器

通用处理器 (GPP) 并不针对特定的应用领域进行体系结构和指令集的优化, 它们具有一般化的通用体系结构和指令集, 以求支持复杂的运算并易于添加新开发的功能。一般而言, 在嵌入式微控制器 (MCU) 和微处理器 (MPU) 中会包含一个通用处理器核。

MPU 通常代表一个 CPU (中央处理器), 而 MCU 则强调把中央处理器、存储器和外围电路集成在一个芯片中。早期, 微控制器被称为单片机, 指把计算机集成在一个芯片内。嵌入式微控制器也常被称作片上系统 (SoC), 含义是在一个芯片上设计了整个系统。芯片厂商在推出 MCU 时, 往往会有明确的市场定位, 如定位于 PDA、MP3、ADSL 等。定位不同的产品可能包含共同的 CPU 核, 但是集成的扩展电路则不一样。图 2.1 所示给出了一个典型的集成了外围电路的 MCU 的结构。

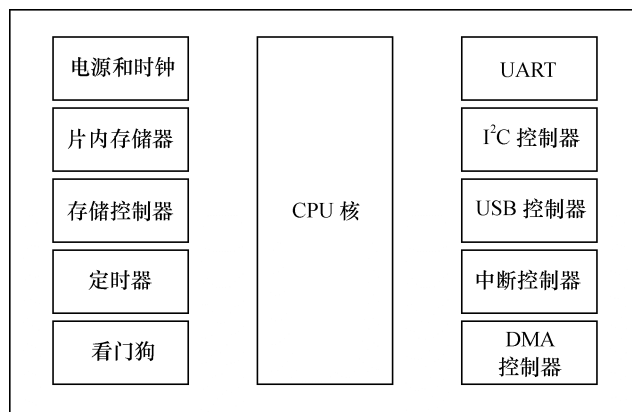


图 2.1 典型的 MCU 内部结构

举个例子, Intel 的 80386 属于微处理器, 而内部集成了 80386 处理器、片选单元、中断控制、定时器、看门狗定时器、串行 I/O、DMA 和总线仲裁、DRAM 控制器等的 386EX 则是 80386 微处理器的微控制器版本。但是, 要说明的是, GPP、MCU 和 MPU 等概念其实非常含糊, 许多地方并不加以区分, 而明确区分这些概念在技术上本身也没有太大的意义。

嵌入式微控制器一般由一个 CPU 核和多个外围电路集成, 目前主流的嵌入式 CPU 核有如下几种。

- Advanced RISC Machines 公司的 ARM。

ARM 内核的设计技术被授权给数百家半导体厂商, 做成不同的 SoC 芯片。ARM 的功耗很低, 在当今最活跃的无线局域网、3G、手机终端、手持设备、有线网络通信设备等中应用非常广泛。本书所基于的 LDD6410 开发板上采用的就是 S3C6410 这个 ARM SoC 芯片。

- MIPS 技术公司的 MIPS。

两个最重要的 MIPS 芯片厂商为 PMC 和 IDT, PMC-Sierra 公司的 MIPS 处理器被 CISCO 公

司大量采用在高端路由器上。IDT 公司在 MIPS 核上集成 PCI 接口，广泛用于以太网交换，另外也尝试增加了 HDLC、Ethernet、串口、SDRAM 控制器、片选、DMA 控制器等外设接口，以用于低端通信产品。

- IBM 和 Motorola 的 PowerPC。

PowerPC 处理器是通信和工控领域应用最广泛的处理器，国内包括华为、中兴在内的通信公司都大量使用 PowerPC，MPC860 和 MPC8260 是其最经典的两款。

- Motorola 公司独有的内核 68K/COLDFIRE。

68K 内核是最早在嵌入式领域广泛应用的内核，其最著名的代表芯片是 68360。Coldfire 则继承了 68K 的特点并对其保持了兼容。Coldfire 内核被用于 DSP 模块、CAN 总线模块以及一般嵌入式处理器所集成的外设模块，在工业控制、机器人研究、家电控制等领域被广泛采用。



Motorola 的半导体部已经独立为飞思卡尔半导体公司 (Freescale Semiconductor Inc.), 因为历史原因, 上文仍然使用 Motorola。

中央处理器的体系架构可以分为两类，一类为冯·诺伊曼结构，一类为哈佛结构。

冯·诺伊曼结构也称普林斯顿结构，是一种将程序指令存储器和数据存储器合并在一起的存储器结构。程序指令存储地址和数据存储地址指向同一个存储器的不同物理位置，因此程序指令和数据的宽度相同。而哈佛结构将程序指令和数据分开存储，指令和数据可以有不同的数据宽度。此外，哈佛结构还采用了独立的程序总线 and 数据总线，分别作为 CPU 与每个存储器之间的专用通信路径，具有较高的执行效率。图 2.2 描述了冯·诺伊曼结构和哈佛结构的区别。

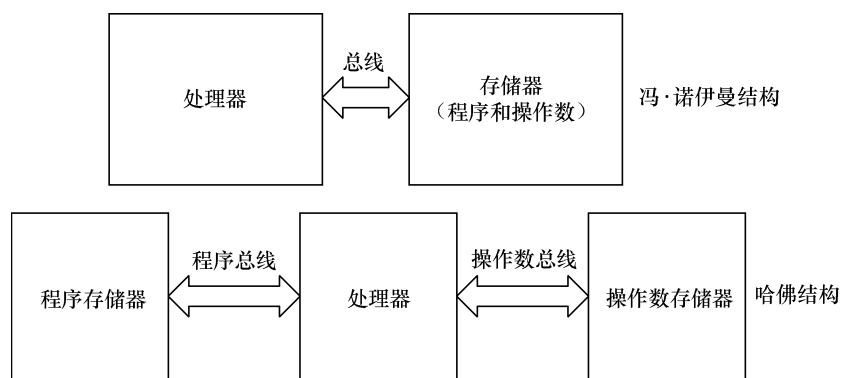


图 2.2 冯·诺伊曼结构与哈佛结构

从指令集的角度来讲，中央处理器也可以分为两类，即 RISC（精简指令集计算机）和 CISC（复杂指令集计算机）。CISC 强调增强指令的能力、减少目标代码的数量，但是指令复杂，指令周期长；而 RISC 强调尽可能减少指令集、指令单周期执行，但是目标代码会更大。ARM、MIPS、PowerPC 等 CPU 内核都采用了 RISC 指令集。目前，RISC 和 CISC 二者的融合非常明显。

2.1.2 数字信号处理器

数字信号处理器（DSP）针对通信、图像、语音和视频处理等领域的算法而设计。它包含独



立的硬件乘法器。DSP 的乘法指令一般在单周期内完成，且优化了卷积、数字滤波、FFT（快速傅立叶变换）、相关、矩阵运算等算法中的大量重复乘法。

DSP 一般采用如图 2.3 所示的改进的哈佛架构，它具有独立的地址总线 and 数据总线，两条总线由程序存储器和数据存储器分时共用。

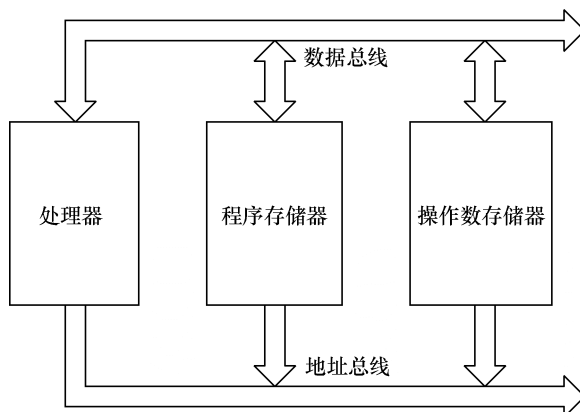


图 2.3 改进的哈佛结构

DSP 分为两类，一类是定点 DSP，一类是浮点 DSP。浮点 DSP 的浮点运算用硬件来实现，可以在单周期内完成，因而其浮点运算处理速度高于定点 DSP。而定点 DSP 只能用定点运算模拟浮点运算。

德州仪器（TI）、美国模拟器件公司（ADI）是全球 DSP 的两大主要厂商。

TI 的 TMS320™DSP 平台包含了功能不同的多个系列如 2000 系列、3000 系列、4000 系列、5000 系列、6000 系列，工程师也习惯称其为 2X、3X、4X、5X、6X。2010 年 5 月，TI 已经宣布为其 C64x 系列数字信号处理器与多核片上系统提供 Linux 内核支持，以充分满足通信与关键任务基础设施、医疗诊断以及高性能测量测试等应用需求。TI 对 C64x Linux 内核的产品支持 TMS320C6474、TMS320C6455 和 TMS320C6457，将于 2010 年第 3 季度开始提供。

ADI 主要有 16 位定点的 21xx 系列、32 位浮点的 SHARC 系列、从 SHARC 系列发展而来的 TigerSHARC 系列及高性能 16 位 DSP 信号处理能力与通用微控制器方便性相结合的 blackfin 系列等。ADI 的 blackfin 不含 MMU，完整支持 Linux，是 MMU-less 情况下 Linux 的典型示例，其官方网站为 <http://blackfin.uclinux.org/gf/>，目前 blackfin 的 Linux 开发保持了 Linux mainline 的同步。

通用处理器和数字信号处理器也有相互融合以取长补短的趋势，如数字信号控制器（DSC）即为 MCU+DSP，ADI 的 blackfin 系列就属于 DSC。目前，芯片厂商也推出了许多 ARM+DSP 的双核以及多核的处理器，如 TI 公司的 OMAP 4 平台就包括 4 个主要处理引擎：ARM Cortex-A9 MPCore、PowerVR SGX 540 GPU（Graphic Processing Unit）、C64x DSP 和 ISP（Image Signal Processor）。

除了上面讲述的通用微控制器和数字信号处理器外，还有一些针对特定领域而设计的专用处理器（ASP），它们都是针对一些特定应用而设计的，如用于 HDTV、ADSL、Cable Modem 等的专用处理器。

网络处理器是一种可编程器件，它应用于电信领域的各种任务，如包处理、协议分析、路由

查找、声音/数据的汇聚、防火墙、QoS 等。网络处理器器件内部通常由若干个微码处理器和若干硬件协处理器组成，多个微码处理器在网络处理器内部并行处理，通过预先编制的微码来控制处理流程。而对于一些复杂的标准操作（如内存操作、路由表查找算法、QoS 的拥塞控制算法、流量调度算法等）则采用硬件协处理器来进一步提高处理性能，从而实现了业务灵活性和高性能的有机结合。

对于某些应用场合，使用 ASIC（专用集成电路）往往是低成本且高性能的方案。ASIC 专门针对特定应用而设计，不具备也不需要灵活的编程能力。使用 ASIC 完成同样的功能往往比直接使用 CPU 资源或 CPLD（复杂可编程逻辑器件）/FPGA（现场可编程门阵列）来得更廉价且高效。

在实际项目的硬件方案中，往往会根据应用的需求选择通用处理器、数字信号处理器、特定领域处理器、CPLD/FPGA 或 ASIC 之一的解决方案，在复杂的系统中，这些芯片可能会同时存在，协同合作，各自发挥自己的长处。如在一款智能手机中，可使用 MCU 处理图形用户界面和用户的按键输入并运行多任务操作系统，使用 DSP 进行音视频编解码，而在射频方面则采用 ASIC。

综合 2.1 节的内容，我们可得出如图 2.4 所示的处理器分类。

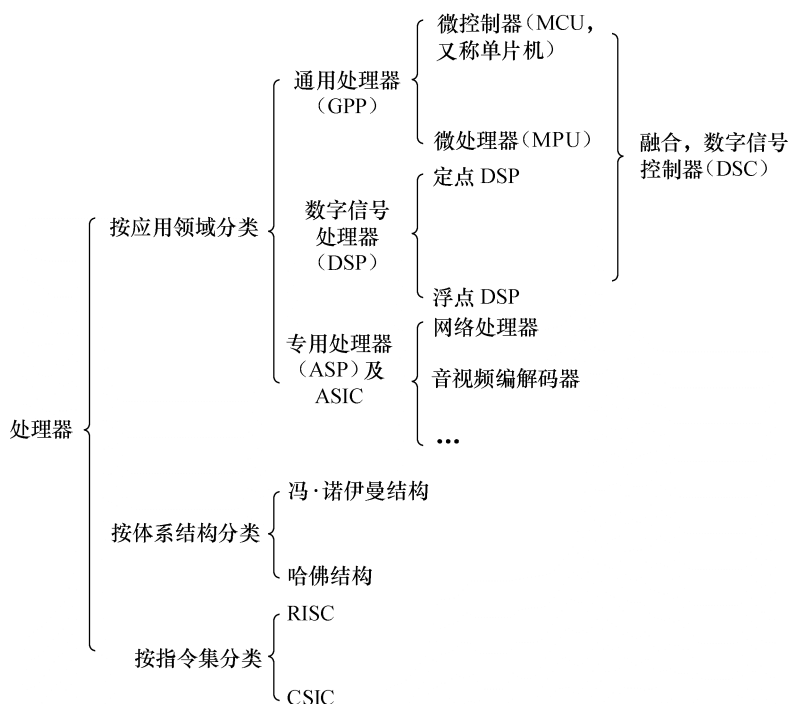


图 2.4 处理器分类

2.2 存储器

存储器主要可分类为只读存储器（ROM）、闪存（Flash）、随机存取存储器（RAM）、光、



磁介质存储器。

ROM 还可再细分为不可编程 ROM、可编程 ROM (PROM)、可擦除可编程 ROM (EPROM) 和电可擦除可编程 ROM (EEPROM)，EEPROM 完全可以用软件来擦写，已经非常方便了。

目前 ROM 有被 Flash 替代的趋势，NOR（或非）和 NAND（与非）是市场上两种主要的 Flash 闪存技术。Intel 于 1988 年首先开发出 NOR Flash 技术，彻底改变了原先由 EPROM 和 EEPROM 一统天下的局面。紧接着，1989 年，东芝公司发表了 NAND Flash 结构，每比特的成本被大大降低。

NOR Flash 和 CPU 的接口属于典型的类 SRAM 接口（如图 2.5 所示），不需要增加额外的控制电路。NOR Flash 的特点是可芯片内执行（XIP，eXecute In Place），程序可以直接在 NOR 内运行。而 NAND Flash 和 CPU 的接口必须由相应的控制电路进行转换，当然也可以通过地址线或 GPIO 产生 NAND Flash 接口的信号。NAND FLASH 以块方式进行访问，不支持芯片内执行。

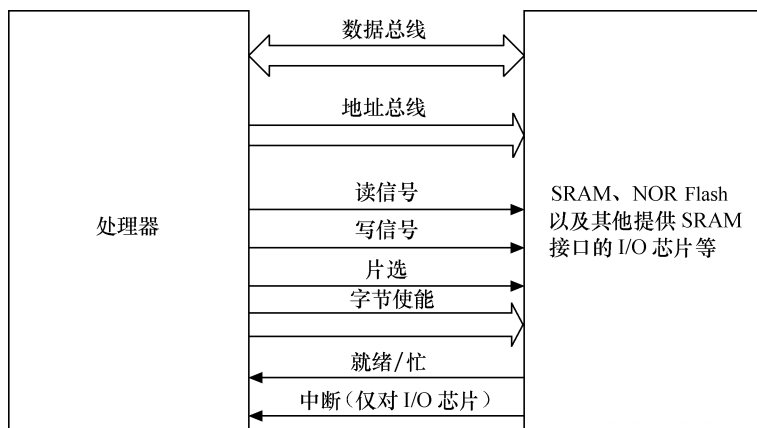


图 2.5 典型的类 SRAM 接口

公共闪存接口（Common Flash Interface，简称 CFI）是一个公开的、标准的从 NOR Flash 器件中读取数据的接口。它可以使系统软件查询已安装的 Flash 器件的各种参数，包括器件阵列结构参数、电气和时间参数以及器件支持的功能等。利用 CFI，在不修改系统软件的情况下，就可以用新型的和改进的产品代替旧版本的产品。

一个 NAND Flash 的接口主要包含如下信号。

- I/O 总线：地址、指令和数据通过这组总线传输，一般为 8 位或 16 位。
- 芯片启动（Chip Enable，CE#）：如果没有检测到 CE 信号，那么，NAND 器件就保持待机模式，不对任何控制信号做出响应。
- 写使能（Write Enable，WE#）：WE# 负责将数据、地址或指令写入 NAND 之中。
- 读使能（Read Enable，RE#）：RE# 允许数据输出。
- 指令锁存使能（Command Latch Enable，CLE）：当 CLE 为高时，在 WE# 信号的上升沿，指令将被锁存到 NAND 指令寄存器中。
- 地址锁存使能（Address Latch Enable，ALE）：当 ALE 为高时，在 WE# 信号的上升沿，

地址将被锁存到 NAND 地址寄存器中。

- 就绪/忙 (Ready/Busy, R/B#): 如果 NAND 器件忙, R/B#信号将变低。该信号是漏极开路, 需要采用上拉电阻。

NAND Flash 较 NOR Flash 容量大, 价格低; NAND Flash 中每个块的最大擦写次数是一百万次, 而 NOR 的擦写次数是十万次; NAND Flash 的擦除、编程速度远超过 NOR Flash。

由于 Flash 固有的电器特性, 在读写数据过程中, 偶尔会产生 1 位或几位数据错误, 即位反转, NAND Flash 发生位反转的几率要远大于 NOR Flash。位反转无法避免, 因此, 使用 NAND Flash 的同时, 应采用错误探测/错误更正 (EDC/ECC) 算法。

Flash 的编程原理都是只能将 1 写为 0, 而不能将 0 写为 1。所以在 Flash 编程之前, 必须将对应的块擦除, 而擦除的过程就是把所有位都写为 1 的过程, 块内的所有字节变为 0xFF。

许多嵌入式系统都提供了 IDE (Integrated Drive Electronics) 接口, 以供连接硬盘控制器或光驱, IDE 接口的信号与 SRAM 类似。人们通常也把 IDE 接口称为 ATA (Advanced Technology Attachment) 接口, 技术角度而言并不准确。其实, ATA 接口发展至今, 已经经历了 ATA-1 (IDE)、ATA-2 (EIDE Enhanced IDE/Fast ATA)、ATA-3 (FastATA-2)、Ultra ATA、Ultra ATA/33、Ultra ATA/66、Ultra ATA/100 及 Serial ATA 的发展过程。

以上所述的各种 ROM、Flash 和磁介质存储器都属于非易失性存储器 (NVM) 的范畴, 掉电信息不会丢失, 而 RAM 则与此相反。

RAM 也可再分为静态 RAM (SRAM) 和动态 RAM (DRAM)。DRAM 以电荷形式进行存储, 数据存储在电容器中。由于电容器会由于漏电而导致电荷丢失, 因而 DRAM 器件需要定期被刷新。SRAM 是静态的, 只要供电它就会保持一个值, SRAM 没有刷新周期。每个 SRAM 存储单元由 6 个晶体管组成, 而 DRAM 存储单元由 1 个晶体管和 1 个电容器组成。

通常所说的 SDRAM、DDR SDRAM 皆属于 DRAM 的范畴, 它们采用与 CPU 外存控制器同步的时钟工作 (注意不是与 CPU 的工作频率一致)。与 SDRAM 相比, DDR SDRAM 同时利用了时钟脉冲的上升沿和下降沿传输数据, 因此在时钟频率不变的情况下, 数据传输频率被加倍。此外, 还存在使用 RSL (Rambus 发信电平) 技术的 RDRAM (Rambus DRAM) 和 Direct RDRAM。

针对许多特定场合的应用, 嵌入式系统中往往还使用了一些特定类型的 RAM。

1. NVRAM: 非易失性 RAM

既然是 RAM, 就是易失性的, 为什么会有一类非易失性的 RAM 呢?

实际上, NVRAM 借助带有备用电源的 SRAM 或借助 NVM (如 EEPROM) 存储 SRAM 的信息并恢复来实现。NVRAM 的特点是完全像 SRAM 一样读写, 而且写入的信息掉电不丢失, 不需要 EEPROM 和 Flash 的特定擦除和编程操作。NVRAM 多用于存放系统中的参数信息。

2. DPRAM: 双端口 RAM

DPRAM 的特点是可以同时通过 2 个端口同时访问, 具有 2 套完全独立的数据总线、地址总线线和读写控制线, 通常用于 2 个处理器之间交互数据, 如图 2.6 所示。当一端被写入数据后, 另一端可以通过轮询或中断获知, 并读取其写入的数据。由于双 CPU 同时访问 DPRAM 时的仲裁逻辑电路集成在 DPRAM 内部, 因而硬件工程师设计电路的原理比较简单。

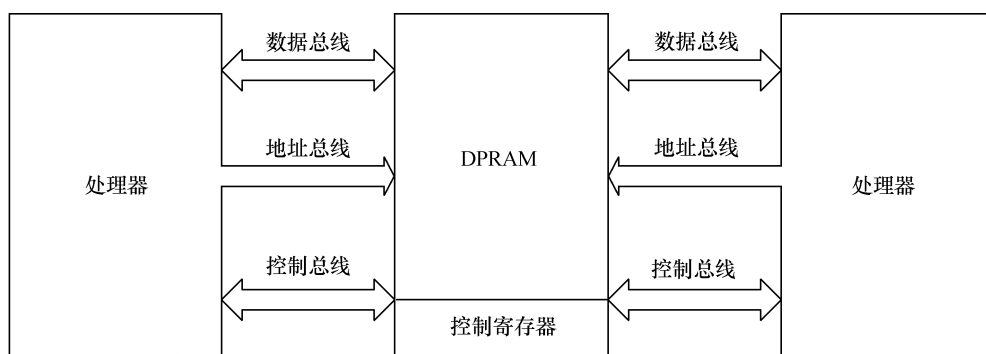


图 2.6 双端口 RAM

DPRAM 的优点是通信速度快、实时性强、接口简单,而且两边 CPU 都可主动进行数据传输。除了双端口 RAM 以外,目前 IDT 等芯片厂商还推出了多端口 RAM,可以供 3 个以上的 CPU 互通数据。

3. CAM: 内容寻址 RAM

CAM 是以内容进行寻址的存储器,是一种特殊的存储阵列 RAM,它的主要工作机制就是将一个输入数据项与存储在 CAM 中的所有数据项自动同时进行比较,判别该输入数据项与 CAM 中存储的数据项是否相匹配,并输出该数据项对应的匹配信息。

如图 2.7 所示,在 CAM 中,输入的是所要查询的数据,输出的是数据地址和匹配标志。若匹配(即搜寻到数据),则输出数据地址。CAM 用于数据检索的优势是软件无法比拟的,可以极大地提高系统性能。

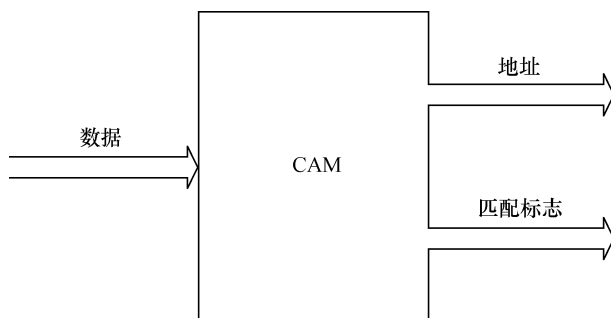


图 2.7 CAM 的输入与输出

● FIFO: 先进先出队列

FIFO 存储器的特点是先进先出,进出有序,FIFO 多用于数据缓冲。FIFO 和 DPRAM 类似,具有两个访问端口,但是 FIFO 两边的端口并不对等,某一时刻只能被设置为一边作为输入,一边作为输出。

如果 FIFO 的区域共为 n 个字节,我们只能通过循环 n 次读取同一个地址才能将该片区域读出,不能指定偏移地址。对于有 n 个数据的 FIFO,当循环读取 m 次,下一次读会自动读取到第 $m+1$ 个数据,这是由 FIFO 本身的特性决定的。

总结 2.2 节的内容,我们可得出如图 2.8 所示的存储器分类。

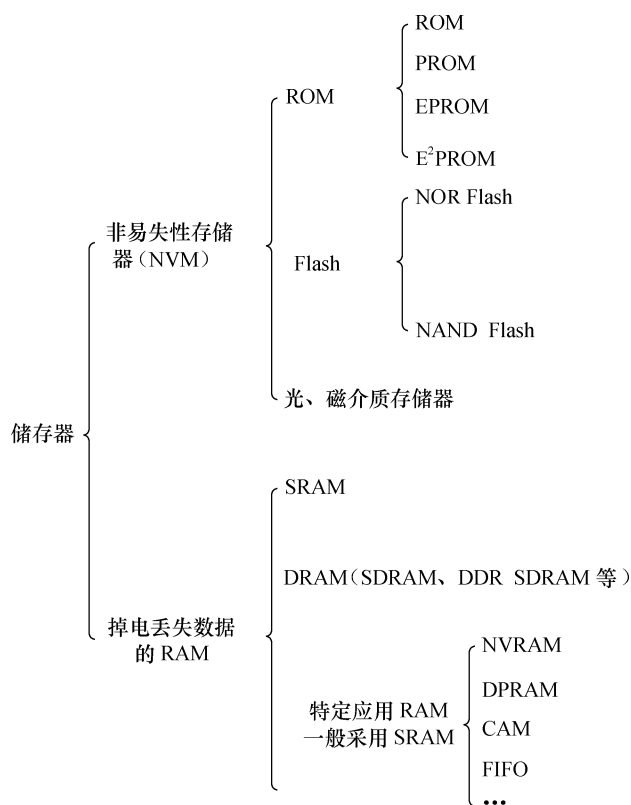


图 2.8 存储器分类

2.3 接口与总线

2.3.1 串口

RS-232、RS-422 与 RS-485 都是串行数据接口标准，最初都是由电子工业协会（EIA）制订并发布的。

RS-232 在 1962 年发布，命名为 EIA-232-E。之后发布的 RS-422 定义了一种平衡通信接口，它是一种单机发送、多机接收的单向、平衡传输规范，被命名为 TIA/EIA-422-A 标准。RS-422 改进了 RS-232 通信距离短、速率低的缺点。为进一步扩展应用范围，EIA 又于 1983 年在 RS-422 的基础上制定了 RS-485 标准，增加了多点、双向通信能力，即允许多个发送器连接到同一条总线上，同时增加了发送器的驱动能力和冲突保护特性，并扩展了总线共模范围，被命名为 TIA/EIA-485-A 标准。

1969 年发布的 RS-232 修改版 RS-232C 是嵌入式系统应用最广泛的串行接口，它为连接 DTE（数据终端设备）与 DCE（数据通信设备）而制定。RS-232C 规标准接口有 25 条线（4 条数据线、11 条控制线、3 条定时线、7 条备用和未定义线），常用的只有 9 根，它们是 RTS/CTS（请求发送/



清除发送流控制)、RxD/TxD (数据收发)、DSR/DTR (数据终端就绪/数据设置就绪流控制)、DCD (数据载波检测, 也称 RLSD, 即接收线信号检出)、Ringin-RI (振铃指示)、SG (信号地) 信号。RTS/CTS、TxD/RxD、DRS/DTR 等信号的定义如下。

- RTS: 用来表示 DTE 请求 DCE 发送数据, 当终端要发送数据时, 使该信号有效。
- CTS: 用来表示 DCE 准备好接收 DTE 发来的数据, 是对 RTS 的响应信号。
- TxD: DTE 通过 TxD 将串行数据发送到 DCE。
- RxD: DTE 通过 RxD 接收从 DCE 发来的串行数据。
- DSR: 有效 (ON 状态) 表明 DCE 可以使用。
- DTR: 有效 (ON 状态) 表明 DTE 可以使用。
- DCD: 当本地 DCE 设备收到对方 DCE 设备送来的载波信号时, 使 DCD 有效, 通知 DTE 准备接收, 并且由 DCE 将接收到的载波信号解调为数字信号, 经 RXD 线送给 DTE。
- Ringin-RI: 当 MODEM 收到交换台送来的振铃呼叫信号时, 使该信号有效 (ON 状态), 通知终端, 已被呼叫。

在嵌入式系统中, 并不太注重 DTE 和 DCE 的概念, 而 RS-232C 也很少用来连接 modem, 多使用 RS-232C 进行对等通信, 如 Windows 超级终端、Linux minicom 用来连接电路板控制台等。最简单的 RS-232C 串口只需要连接 RxD、TxD、SG 这 3 个信号, 使用 XON/XOFF 软件流控。

组成一个 RS-232C 串口的硬件原理如图 2.9 所示, 从 CPU 到连接器依次为: CPU、UART (通用异步接收器发送器, 作用是完成并/串转换)、CMOS/TTL 电平与 RS-232C 电平转换、DB9/DB25 或自定义连接器。

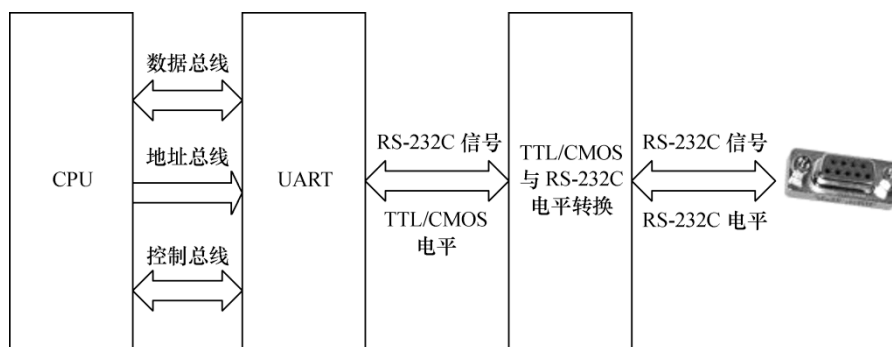


图 2.9 RS-232C 串口电路原理

2.3.2 I²C

I²C (内置集成电路) 总线是由 Philips 公司开发的两线式串行总线, 产生于 20 世纪 80 年代, 用于连接微控制器及其外围设备。I²C 总线简单而有效, 占用很少的 PCB (印刷电路板) 空间, 芯片管脚数量少, 设计成本低。I²C 总线支持多主控 (multi-mastering) 模式, 任何能够进行发送和接收的设备都可以成为主设备。主控能够控制数据的传输和时钟频率, 在任意时刻只能有一个主控。

组成 I²C 总线的两个信号为数据线 SDA 和时钟 SCL。为了避免总线信号的混乱, 要求各设备连接到总线的输出端必须是开漏输出或集电极开路输出的结构。总线空闲时, 上拉电阻使 SDA 和 SCL 线都保持高电平。根据开漏输出或集电极开路输出信号的“线与”逻辑, I²C 总线上任意

器件输出低电平都会使相应总线上的信号线变低。



“线与”逻辑指的是两个或两个以上的输出直接互连就可以实现“AND”的逻辑功能，只有输出端是开漏（对于 CMOS 器件）输出或集电极开路（对于 TTL 器件）输出时才满足此条件。工程师一般以“OC 门”简称开漏或集电极开路。

I²C 设备上的串行数据线 SDA 接口电路是双向的，输出电路用于向总线上发送数据，输入电路用于接收总线上的数据。同样地，串行时钟线 SCL 也是双向的，作为控制总线数据传送的主机要通过 SCL 输出电路发送时钟信号，并检测总线上 SCL 上的电平以决定什么时候发下一个时钟脉冲电平；作为接收主机命令的从设备需按总线上 SCL 的信号发送或接收 SDA 上的信号，它也可以向 SCL 线发出低电平信号以延长总线时钟信号周期。

当 SCL 稳定在高电平时，SDA 由高到低的变化将产生一个开始位，而由低到高的变化则产生一个停止位，如图 2.10 所示。

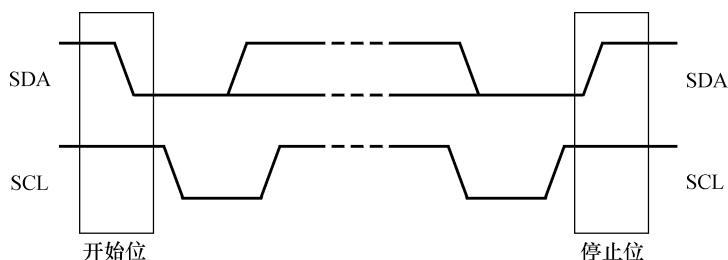


图 2.10 I²C 总线开始位和停止位

开始位和停止位都由 I²C 主设备产生。在选择从设备时，如果从设备采用 7 位地址，则主设备在发起传输过程前，需先发送 1 字节的地址信息，前 7 位为设备地址，最后 1 位为读写标志。之后，每次传输的数据也是 1 个字节，从 MSB 位开始传输。每个字节传完后，在 SCL 的第 9 个上升沿到来之前，接收方应该发出 1 个 ACK 位。SCL 上的时钟脉冲由 I²C 主控方发出，在第 8 个时钟周期之后，主控方应该释放 SDA，I²C 总线的时序如图 2.11 所示。

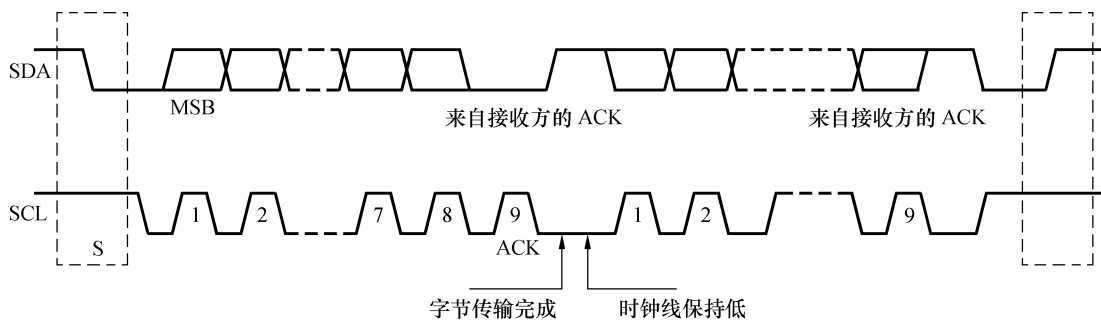


图 2.11 I²C 总线时序

2.3.3 USB

USB（通用串行总线）是 Intel、Microsoft 等厂商为解决计算机外设种类的日益增加与有限的主板插槽和端口之间的矛盾而于 1995 年提出的，它具有数据传输率高、易扩展、支持即插即用和



热插拔的优点, 目前已得到广泛应用

USB 1.1 包含全速和低速两种模式, 低速方式的速率为 1.5Mbit/s, 支持一些不需要很大数据吞吐量和很高实时性的设备, 如鼠标等。全速模式为 12Mbit/s, 可以外接速率更高的外设。在 USB 2.0 中, 增加了一种高速方式, 数据传输率达到 480Mbit/s, 可以满足更高速外设的需要。

USB 的物理拓扑结构如图 2.12 所示, 在 USB 2.0 中, 高速方式下 Hub 使全速和低速方式的信令环境独立出来, 图 2.13 所示高速方式下 Hub 的作用。

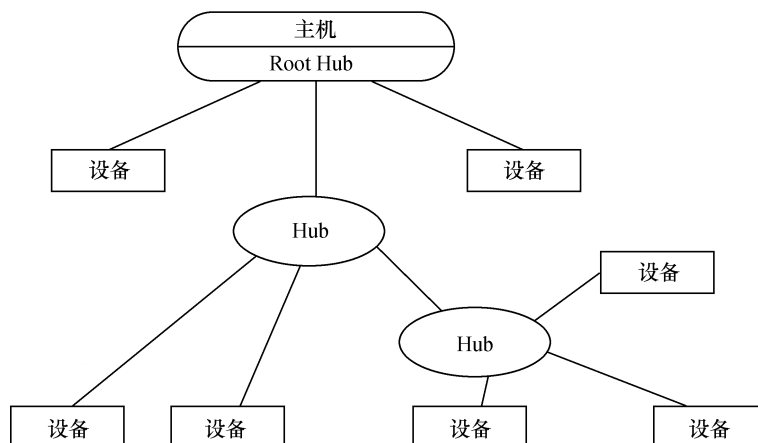


图 2.12 USB 的物理拓扑

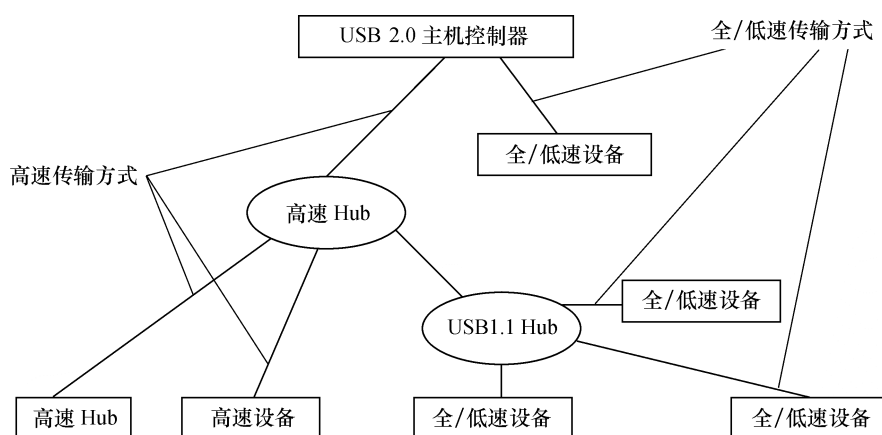


图 2.13 USB 2.0 连接高速、全/低速设备

在嵌入式系统中, 电路板若需要挂接 USB 设备 (device), 则需提供 USB 主机 (host) 控制器和连接器; 若电路板需要作为 USB 设备, 则需提供 USB 设备适配器和连接器。有的 MCU 集成了 USB 主机控制器和设备适配器。

USB 总线的机械连接非常简单, 采用 4 芯的屏蔽线, 一对差分线 (D+, D-) 传送信号, 另一对 (VBUS, 电源地) 传送 +5V 的直流电。一个 USB 主控制器端口最多可连接 127 个器件, 各器件之间的距离不超过 5 米。

USB 提供了 4 种传输方式以适应各种设备的需要, 说明如下。

(1) 控制 (Control) 传输方式。

控制传输是双向传输，数据量通常较小，主要用来进行查询、配置和给 USB 设备发送通用的命令

(2) 同步 (Synchronization) 传输方式。

同步传输提供了确定的带宽和间隔时间，它被用于时间严格并具有较强的容错性的流数据传输，或者用于要求恒定的数据传送率的即时应用。例如进行语音业务传输时，使用同步传输方式是更好的选择

(3) 中断 (Interrupt) 传输方式。

中断方式传送是单向的，对于 USB 主机而言，只有输入。中断传输方式主要用于定时查询设备是否有中断数据要传送，该传输方式应用在少量的、分散的、不可预测的数据传输场合，键盘、游戏杆和鼠标属于这一类型

(4) 批量 (Bulk) 传输方式。

批量传输主要应用在没有带宽和间隔时间要求的批量数据的传送和接收，它要求保证传输打印机和扫描仪属于这种类型

2.3.4 以太网接口

以太网接口由 MAC (以太网媒体接入控制器) 和 PHY (物理接口收发器) 组成。以太网 MAC 由 IEEE-802.3 以太网标准定义，实现了数据链路层。常用的 MAC 支持 10Mbit/s 或 100Mbit/s 两种速率。PHY 则实现物理层功能，IEEE-802.3 标准定义了以太网 PHY，它符合 IEEE-802.3k 中用于 10BaseT (第 14 条) 和 100BaseTX (第 24 条和第 25 条) 的规范。10BaseT 和 100BaseTX PHY 两种实现的帧格式是一样的，但信令机制不同，而且 10BaseT 采用曼彻斯特编码，100BaseTX 采用 4B/5B 编码。

MAC 和 PHY 之间采用 MII (媒体独立接口) 连接，它是 IEEE-802.3 定义的以太网行业标准，包括 1 个数据接口和 1 个 MAC 和 PHY 之间的管理接口。数据接口包括分别用于发送和接收的两条独立信道，每条信道都有自己的数据、时钟和控制信号，MII 数据接口总共需要 16 个信号。MII 管理接口包含两个信号，一个是时钟信号，另一个是数据信号。通过管理接口，上层能监视和控制 PHY。

组成一个以太网接口的硬件原理如图 2.14 所示，从 CPU 到最终接口依次为：CPU、MAC、PHY、以太网隔离变压器、RJ45 插座。以太网隔离变压器是以太网收发芯片与连接器之间的磁性组件，在其两者之间起着信号传输、阻抗匹配、波形修复、信号杂波抑制和高电压隔离作用。

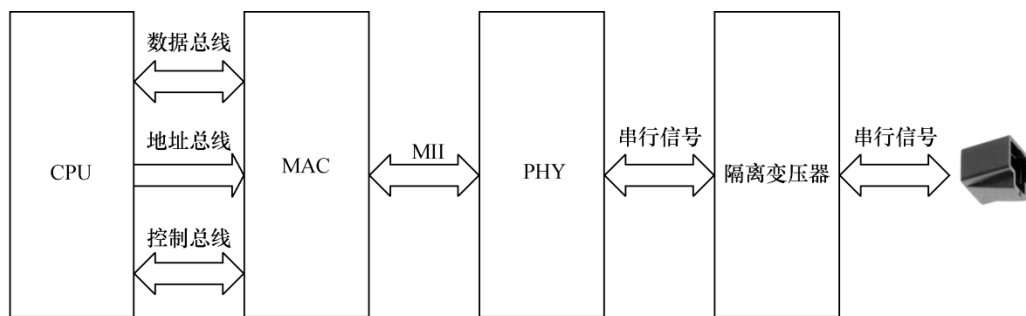


图 2.14 以太网接口电路原理



许多处理器内部集成了 MAC 或同时集成了 MAC 和 PHY，另有许多以太网控制芯片也集成了 MAC 和 PHY。

2.3.5 ISA

ISA（工业标准结构总线）总线起源于 1981 年 IBM 生产的以 Intel 8088 为 CPU 的 IBM-PC 微计算机，开始时总线宽度为 8 位。1984 年推出的 IBM-PC/AT 系统将 ISA 总线扩充为 16 位数据总线宽度，同时地址总线宽度也由 20 位扩充到了 24 位。其后推出的 EISA（扩展的 ISA）采用 32 位地址线，数据总线也扩展为 32 位，但仍保持了与 ISA 的兼容。

图 2.15 所示为 ISA 总线的信号，这些信号可分为 3 组。

- 总线基本信号：ISA 总线工作所需要的最基本信号，含复位、时钟、电源、地等。
- 总线访问信号：用于访问 ISA 总线设备的地址线、数据线以及相应的应答信号。
- 总线控制信号：中断和 DMA 请求。

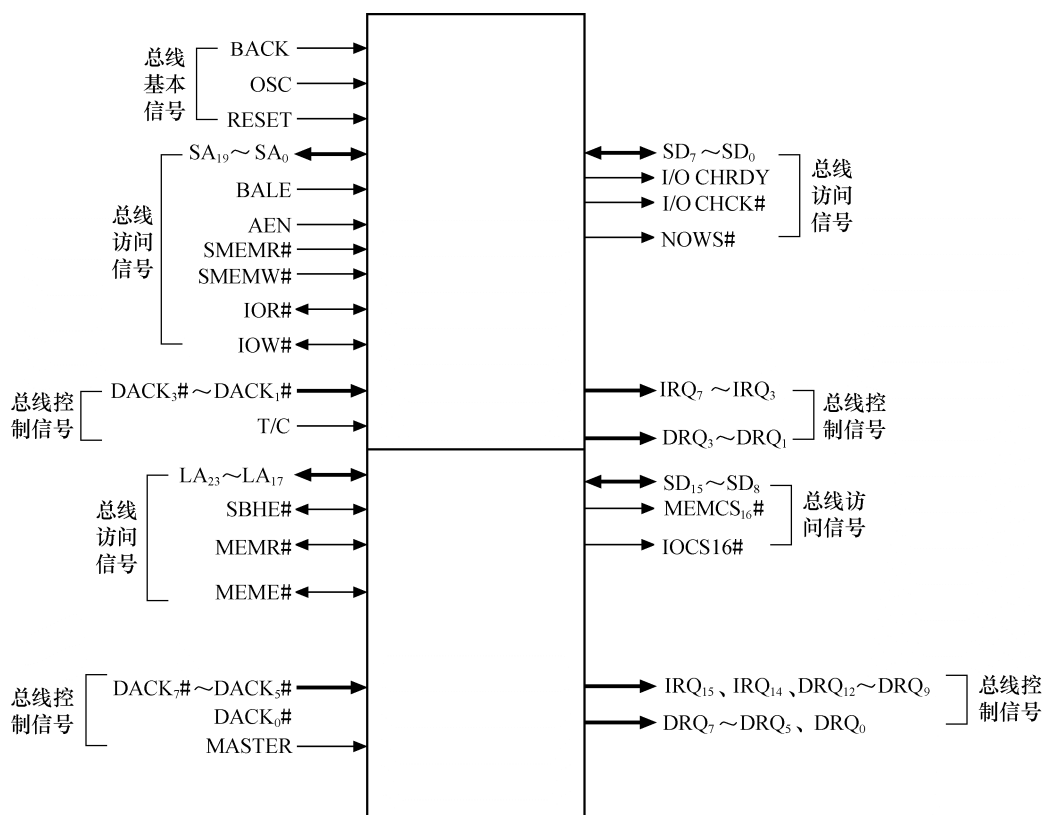


图 2.15 ISA 总线信号

图 2.15 中各信号的详细定义如下。

- RESET、BLCK：复位及总线基本时钟，BLCK 为 8MHz。
- SA₁₉~SA₀：存储器及 I/O 空间 20 位地址，带锁存。
- LA₂₃~LA₁₇：存储器及 I/O 空间 20 位地址，不带锁存。
- BALE：总线地址锁存，外部锁存器的选通。

- AEN: 地址允许, 表明 CPU 让出总线, DMA 开始。
- SMEMR#、SMEMW#: 8 位 ISA 存储器读写控制。
- MEMR#、MEMW#: 16 位 ISA 存储器读写控制。
- SD15~SD0: 数据总线, 访问 8 位 ISA 卡时高 8 位自动传送到 SD7~SD0。
- SBHE#: 高字节允许, 打开 SD15~SD8 数据通路。
- MEMCS16#、IOCS16#: ISA 卡发出此信号确认可以进行 16 位传送。
- I/OCHRDY: ISA 卡准备信号, 可控制插入等待周期。
- NOWS#: 有效则暗示不用插入等待周期。
- I/OCHCK#: ISA 卡奇偶校验错。
- IRQ15、IRQ14、IRQ12~IRQ9、IRQ7~IRQ3: 中断请求。
- DRQ7~DRQ5、DRQ3~DRQ0: ISA 卡 DMA 请求。
- DACK7#~DACK5#、DACK3#~DACK0#: DMA 请求响应。
- MASTER#: ISA 主模块确立信号, ISA 发出此信号, 与主机内 DMAC (DMA 控制器) 配合使 ISA 卡成为主模块。

2.3.6 PCI 和 cPCI

PCI (外围部件互连) 是由 Intel 于 1991 年推出的一种局部总线, 作为一种通用的总线接口标准, 它在目前的计算机系统中得到了非常广泛的应用。PCI 提供了一组完整的总线接口规范, 其目的是描述如何将计算机系统的外围设备以一种结构化和可控化的方式连接在一起, 给出了外围设备在连接时的电气特性和行为规约, 并且详细定义了计算机系统各个不同部件之间应该如何正确地进行交互。PCI 总线具有如下特点。

- 数据总线 32 位, 可扩充到 64 位。
- 可进行突发 (burst) 模式传输。



突发方式传输是指取得总线控制权后连续进行多个数据的传输。突发传输时, 只需要给出目的地的首地址, 访问第 1 个数据后, 第 2~ n 个数据会在首地址基础上按一定规则自动被寻址和传输。与突发方式对应的是单周期方式, 它在 1 个总线周期只传送 1 个数据。

- 总线操作与处理器-存储器子系统操作并行。
- 总线时钟频率为 33MHz 或 66MHz, 最高传输率可达 528MB/s。
- 采用中央集中式总线仲裁。
- 支持全自动配置、资源分配, PCI 卡内有设备信息寄存器组为系统提供卡的信息, 可实现即插即用。
- PCI 总线规范独立于微处理器, 通用性好。
- PCI 设备可以完全作为主控设备控制总线。

图 2.16 给出了一个典型的基于 PCI 总线的计算机系统逻辑示意图, 系统的各个部分通过 PCI 总线和 PCI-PCI 桥连接在一起。CPU 和 RAM 通过 PCI 桥连接到 PCI 总线 0 (即主 PCI 总线), 而具有 PCI 接口的显卡则可以直接连接到主 PCI 总线上。PCI-PCI 桥是一个特殊的 PCI 设备, 它负责将 PCI 总线 0 和 PCI 总线 1 (即从 PCI 主线) 连接在一起, 通常 PCI 总线 1 称为 PCI-PCI 桥的



下游 (downstream)，而 PCI 总线 0 则称为 PCI-PCI 桥的上游 (upstream)。为了兼容旧的 ISA 总线标准，PCI 总线还可以通过 PCI-ISA 桥来连接 ISA 总线，从而支持以前的 ISA 设备。

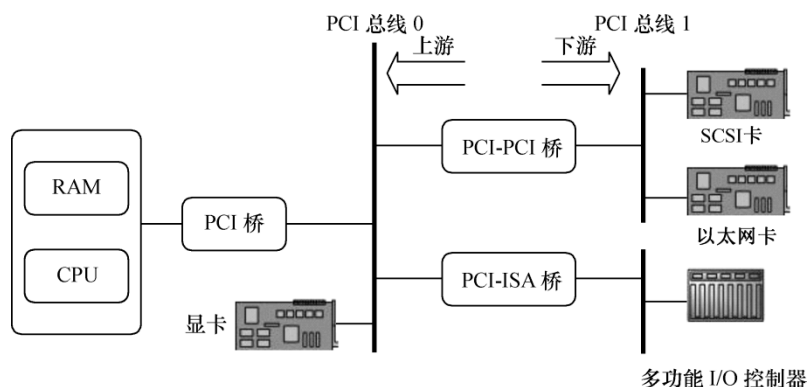


图 2.16 基于 PCI 总线的计算机系统

当 PCI 卡刚加电时，卡上只有配置空间是可被访问的，因而 PCI 卡开始不能由驱动或用户程序访问，这与 ISA 卡有本质的区别（CPU 可直接读取 ISA 卡在存储空间或 I/O 空间映射的地址）。PCI 配置空间保存着该卡工作所需的所有信息，如厂家、卡功能、资源要求、处理能力、功能模块数量、主控卡能力等。通过对这个空间信息的读取与编程，可完成对 PCI 卡的配置。如图 2.17 所示，PCI 配置空间共为 256 字节，主要包括如下信息。

- 制造商标识 (Vendor ID)：由 PCI 组织分配给厂家。
- 设备标识 (Device ID)：按产品分类给本卡的编号。
- 分类码 (Class Code)：本卡功能的分类码，如图卡、显示卡、解压卡等。
- 申请存储器空间：PCI 卡内有存储器或以存储器编址的寄存器和 I/O 空间，为使驱动程序和应用程序能访问它们，需申请 CPU 的一段存储区域以进行定位。配置空间的基地址寄存器用于此目的。
- 申请 I/O 空间：配置空间的基地址寄存器也用来进行系统 I/O 空间的申请。
- 中断资源申请：配置空间中的中断引脚和中断线用来向系统申请中断资源。中断资源的申请通过中断引脚 (interrupt pin) 和中断线 (interrupt line) 来完成的。偏移 3Dh 处为中断引脚寄存器，其值表明 PCI 设备使用了哪一个中断引脚，对应关系为：1-INTA#、2-INTB#、3-INTC#、4-INTD#。

PCI 总线上的信号大体可分为如下几组。

- 系统接口信号。
- 地址与数据接口信号。
- 接口控制信号。
- 仲裁信号。
- 错误报告信号。
- 中断接口信号。
- 其他接口信号。

31		16		15		0		
设备标识				制造商标识				00H
状态				命令				04H
分类码						修正标志		08H
BIST		头类型		延迟定时器		行大小		0CH
基地址寄存器								10H
								14H
								18H
								1CH
								20H
								24H
卡总线 CIS 指针								28H
子系统标识				子系统制造商标识				2CH
扩展 ROM 基地址								30H
保留						容量指针		34H
保留								38H
Max_Lat		Max_Gnt		中断引脚		中断线		3CH

图 2.17 PCI 配置空间

如图 2.18 所示，这些信号的详细定义如下。

- CLK：系统时钟。
- AD31~AD0：地址和数据复用信号线信号。
- C/BE3~C/BE：总线命令和地址使能信号。
- PAR：奇偶校验信号。
- FRAME#：帧周期信号，指示总线操作起始和终止。
- IRDY#：主设备准备好信号。
- TRDY#：目标设备准备好信号。
- STOP#：目标设备要求终止当前数据传输信号。
- DEVSEL#：目标设备选中信号。
- IDSEL：配置空间读写时的片选信号。
- LOCK#：总线锁定信号。
- RST#：复位信号。
- INTA#、INTB#、INTC#和INTD#：中断请求。
- REQ#、GNT#：PCI 总线请求与仲裁后的授权。
- AD63-AD32、C/BE7-4 等：作用于 64 位扩展的 PCI 总线。

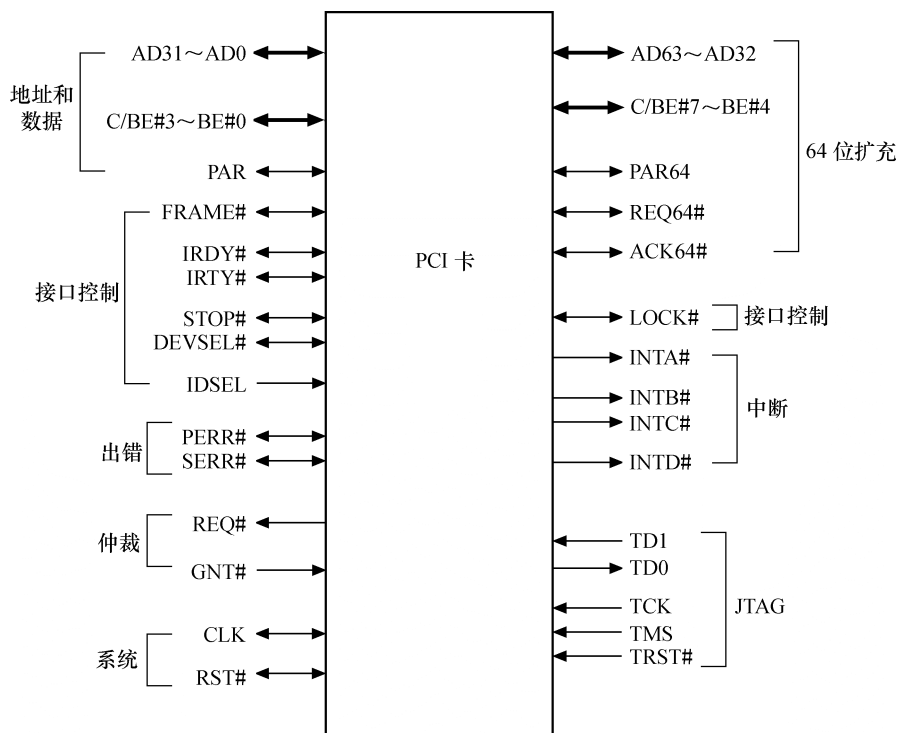


图 2.18 PCI 总线信号

cPCI (Compact PCI, 紧凑型 PCI) 是以 PCI 电气规范为标准的高性能工业用总线, 结合了 VME (Visa Module Eurocard, 维萨信用卡模块欧洲卡) 的高性能、可扩展性和可靠性与 PCI 标准的经济有效和灵活性。cPCI 的 CPU 及外设与标准 PCI 是相同的, 使用与传统 PCI 相同的芯片和软件, 操作系统、驱动和应用程序都感觉不到两者的区别。图 2.19 展示了与 cPCI 总线相关的板卡、背板和机箱, 基本上都是“大块头”, 应用于工业控制和大型通信设备。

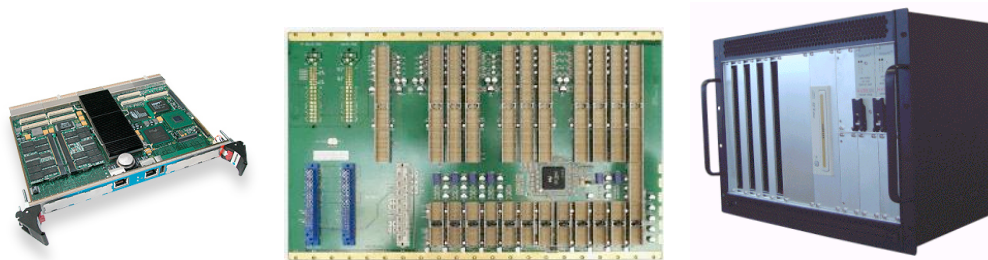


图 2.19 cPCI 板卡、背板与机箱

2.4 CPLD 和 FPGA

CPLD (复杂可编程逻辑器件) 由完全可编程的与或门阵列以及宏单元构成。

CPLD 中基本逻辑单元是宏单元, 宏单元由一些“与或”阵列加上触发器构成, 其中“与或”

阵列完成组合逻辑功能，触发器完成时序逻辑。宏单元中与阵列的输出称为乘积项，其数量标志了 CPLD 的容量。乘积项阵列实际上就是一个“与或”阵列，每一个交叉点都是一个可编程熔丝，如果导通就是实现“与”逻辑。在“与”阵列后一般还有一个“或”阵列，用以完成最小逻辑表达式中的“或”关系。图 2.20 所示为非常典型的 CPLD 的单个宏单元的结构。

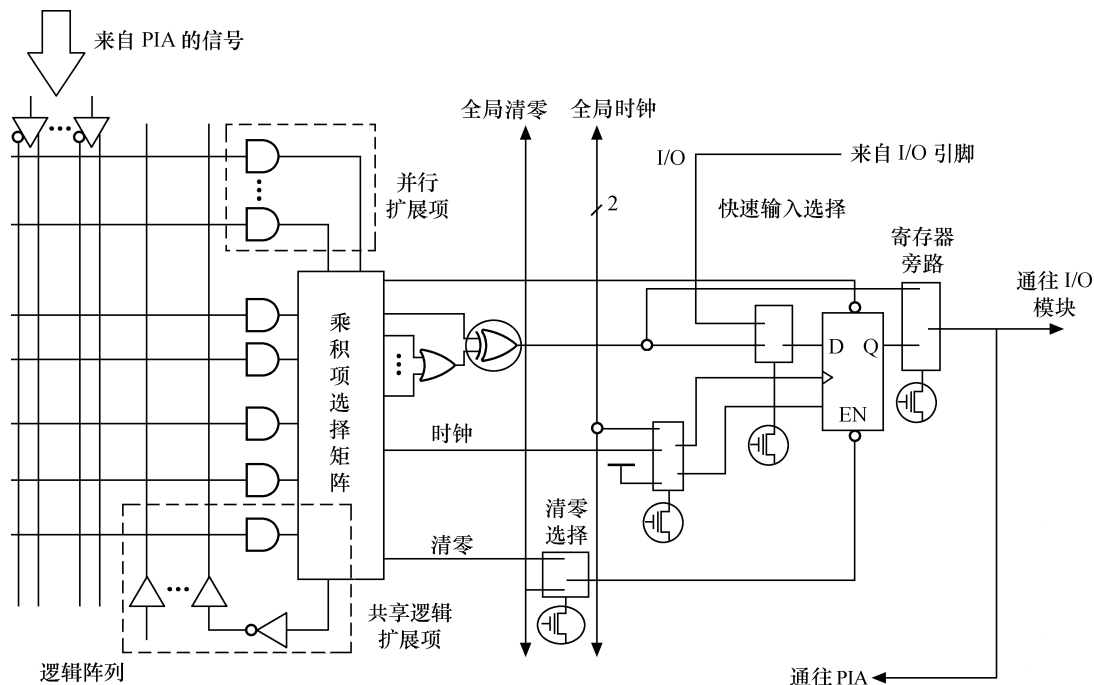


图 2.20 典型的 CPLD 宏单元

图 2.21 给出了一个典型 CPLD 的整体结构。这个 CPLD 由 LAB（逻辑阵列模块，由多个宏单元组成）通过 PIA（可编程互连阵列）互连组成，而 CPLD 与外部的接口则由 I/O 控制模块提供。

图中宏单元的输出会经 I/O 控制块送至 I/O 引脚，I/O 控制块控制每一个 I/O 引脚的工作模式，决定其为输入、输出还是双向引脚，并决定其三态输出的使能端控制。

与 CPLD 不同，FPGA（现场可编程门阵列）基于 LUT（查找表）工艺。查找表本质上是一片 RAM，当用户通过原理图或 HDL（硬件描述语言）语言描述了一个逻辑电路以后，FPGA 开发软件会自动计算逻辑电路的所有可能的结果，并把结果事先写入 RAM。这样，输入一组信号进行逻辑运算就等于输入一个地址进行查表输出对应地址的内容。

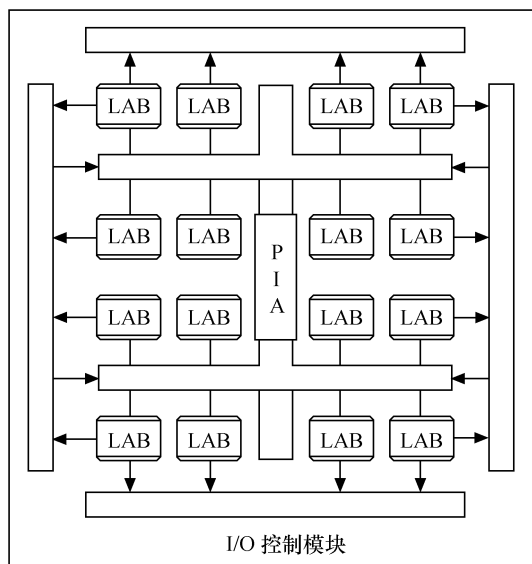


图 2.21 典型 CPLD 的结构



图 2.22 所示为一个典型的 FPGA 的内部结构。这个 FPGA 由 IOC（输入/输出控制模块）、EAB（嵌入式阵列块）、LAB 和 FAST TRACK（快速通道互连）构成。

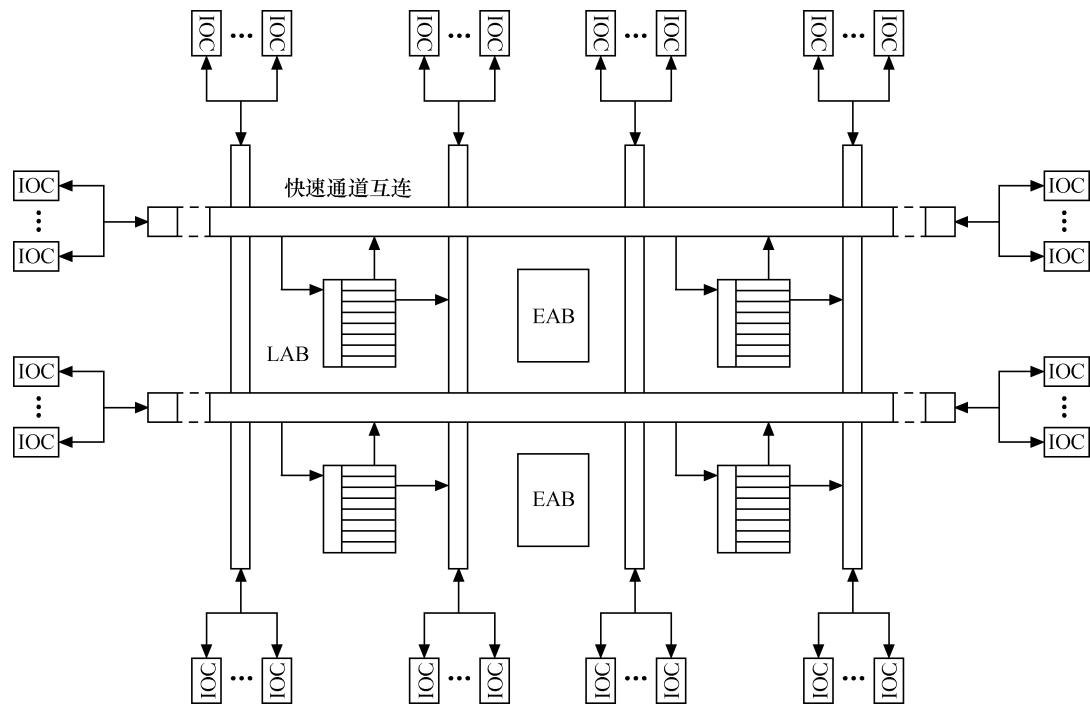


图 2.22 典型 FPGA 的结构

IOC 是内部信号到 I/O 引脚的接口，它位于 FAST TRACK 的行和列的末端，每个 IOC 包含一个双向 I/O 缓冲器和一个既可做输入寄存器也可做输出寄存器的触发器。

EAB（嵌入式存储块）是一种输入输出端带有寄存器的非常灵活的 RAM。EAB 不仅可以用作存储器，也可以事先被写入查表值来以用来构成如乘法器、纠错逻辑等电路。当用于 RAM 时，EAB 可配制成 8 位、4 位、2 位和 1 位长度的数据格式。

LAB 主要用于逻辑电路设计，一个 LAB 包括多个 LE（逻辑单元），每个 LE 包括组合逻辑及一个可编程触发器。一系列 LAB 构成的逻辑阵列用来实现普通逻辑功能，如计数器、加法器、状态机等。

器件内部信号的互连和器件引出端之间的信号互连由 FAST TRACK 连线提供，FAST TRACK 遍布于整个 FPGA 器件，是一系列水平和垂直走向的连续式布线通道。

表 2.1 所示为一个 4 输入 LUT 的实际逻辑电路与 LUT 实现方式的对应关系。

实际逻辑电路与查找表的实现	
实际逻辑电路	LUT 的实现方式

续表

a, b, c, d 输入	逻辑输出	地 址	RAM 中存储的内容
0000	0	0000	0
0001	0	0001	0
....	0	...	0
1111	1	1111	1

CPLD 和 FPGA 的主要厂商有 Altera、Xilinx 和 Lattice 等，采用专门的开发流程，在设计阶段使用 HDL 语言（如 VHDL、Verilog HDL）编程。它们可以实现许多复杂的功能，如实现 UART、I²C 等 I/O 控制芯片、通信算法、音视频编解码算法等，甚至还可以直接集成 ARM 等 CPU 核和外围电路。

对于驱动工程师而言，我们只需要这样看待 CPLD 和 FPGA：如果它完成的是特定的接口和控制功能，我们就直接把它当成由很多逻辑门（与、非、或、D 触发器）组成的完成一系列时序逻辑和组合逻辑的 ASIC；如果它完成的是 CPU 的功能，我们就直接把它当成 CPU。驱动工程师眼里的硬件比 IC 设计师要宏观。

2.5 原理图分析

2.5.1 原理图分析的内容

原理图分析的含义是指通过阅读电路板的原理图获得各种存储器、外设所使用的硬件资源，主要包括存储器和外设控制芯片所使用的片选、中断和 DMA 资源。通过分析片选得出芯片的内存、I/O 基地址，通过分析中断、DMA 信号获得芯片使用的中断号和 DMA 通道，并归纳出如表 2.2 所示的表格。

表 2.2 存储器、外设控制器资源占用表

芯 片	片 选	基 地 址	字 长	大 小	中断号	DMA 通道
DDR SDRAM	Xm1_CS1	0xC0000000	32	128MB		
以太网控制器	CS1	0x18000000 0x18000004	16	4bytes		
.....	

上述表格对驱动开发的意义很重大，实际上，在大多数情况下，硬件工程师已经给驱动工程师提供了这个表格。

2.5.2 原理图的分析方法

原理图的分析方法是以 CPU 为中心向存储器和外设辐射，步骤如下。

(1) 阅读 CPU 部分，获知 CPU 的哪些片选、中断和集成的外设控制器被使用，列出这些元



素 a、b、c...

CPU 引脚比较多时,芯片可能会被分成几个模块单独被画在原理图的不同页上,这时候应该把相应的部分都分析到位。

(2) 对第一步中列出的元素从原理图中对应的外设和存储器电路中分析出实际的使用情况。

硬件原理图中包含如下元素。

- 符号 (symbol)。

symbol 描述芯片的外围引脚以及引脚的信号,对于复杂的芯片,可能被分割为几个 symbol。在 symbol 中,一般把属于同一个信号群的引脚排列在一起。图 2.23 所示为 NOR Flash AM29LV160DB 和 NAND Flash K9F2G08 的 symbol。

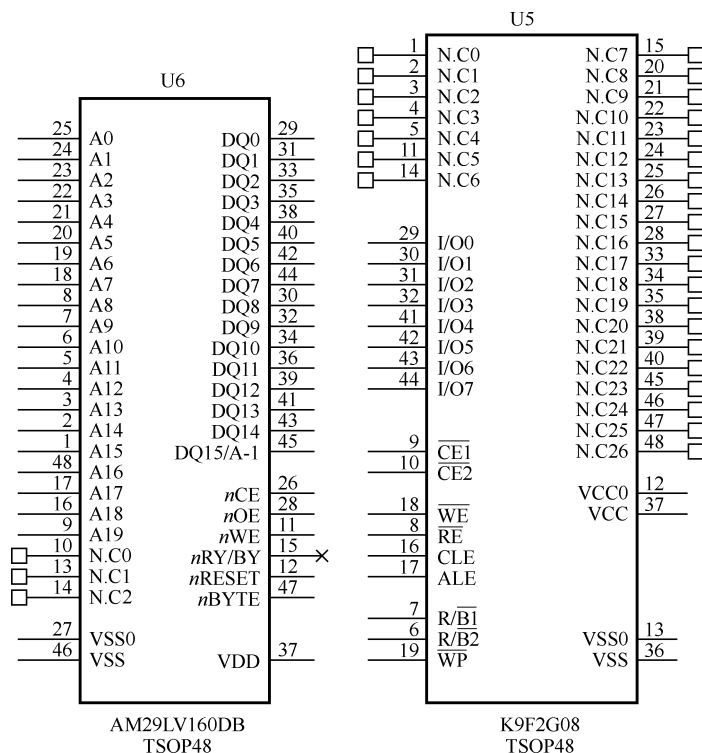


图 2.23 原理图中的 symbol

- 网络 (net)。

描述芯片、接插件和分离元器件引脚之间的互连关系,每个网络需要根据信号的定义赋予一个合适的名字,如果没有给网络取名字,EDA 软件会自动添加一个默认的网络名。添加网络后的 AM29LV160DB 如图 2.24 所示。

- 描述

原理图中会添加一些文字来辅助描述原理图 (类似源代码中的注释),如每页页脚会有该页的功能描述,对重要的信号,在原理图的相应 symbol 和 net 也会附带文字说明。图 2.25 中给出了原理图中描述的例子。

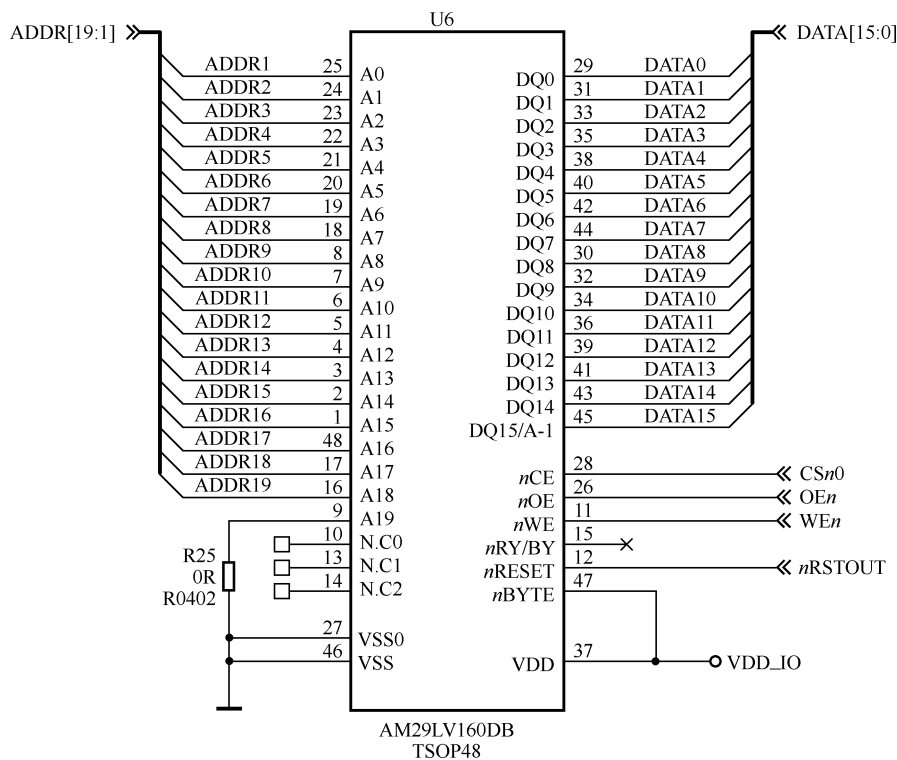


图 2.24 原理图中的 net

LDD6410 Evaluation Board Schematics

Revision	Data	Description
V1	2009.12.26	

图 2.25 原理图中的描述

2.6 硬件时序分析

2.6.1 时序分析的概念

驱动工程师一般不需要分析硬件的时序，但是鉴于许多企业内驱动工程师还需要承担电路板调试的任务，因此，掌握时序分析的方法也就比较必要了。

对驱动工程师或硬件工程师而言，时序分析的意思是让芯片之间的访问满足芯片手册中时序



图信号有效的先后顺序、采样建立时间 (setup time) 和保持时间 (hold time) 的要求, 在电路板工作不正常的时候, 准确地定位时序方面的问题。

建立时间是指在触发器的时钟信号边沿到来以前, 数据已经保持稳定不变的时间, 如果建立时间不够, 数据将不能在这个时钟边沿被打入触发器; 保持时间是指在触发器的时钟信号边沿到来以后, 数据还需稳定不变的时间, 如果保持时间不够, 数据同样不能被打入触发器。如图 2.26 所示, 数据稳定传输必须满足建立和保持时间的要求, 当然在一些情况下, 建立时间和保持时间的值可以为零。

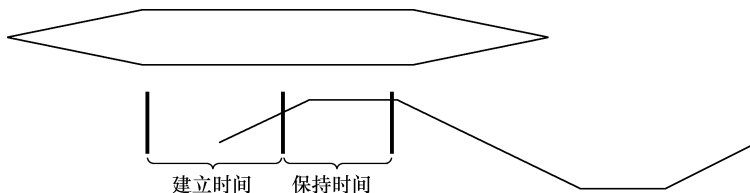


图 2.26 建立时间和保持时间

在工具方面, SynaptiCAD 公司的 Timing Diagrammer Pro 是一种非常好的数字/模拟时序图编辑器及分析引擎。

2.6.2 典型硬件时序

最典型的硬件时序是 SRAM 的读写时序, 在读/写过程中涉及的信号包括地址、数据、片选、读/写、字节使能和就绪/忙。对于一个 16 位、32 位 (甚至 64 位) 的 SRAM, 字节使能表明哪些字节被读写。

图 2.27 给出了 SRAM 的一个读时序, 写时序与此相似。首先是地址总线上输出要读写的地址, 然后 SRAM 片选被发出, 接着读/写信号被输出, 之后读写信号要经历数个等待周期。当 SRAM 读写速度比较慢时, 等待周期可以由 MCU 的相应寄存器设置, 也可以通过设备就绪/忙 (如图 2.27 中的 $nWait$) 向 CPU 报告, 这样, 读写过程中会自动添加等待周期。

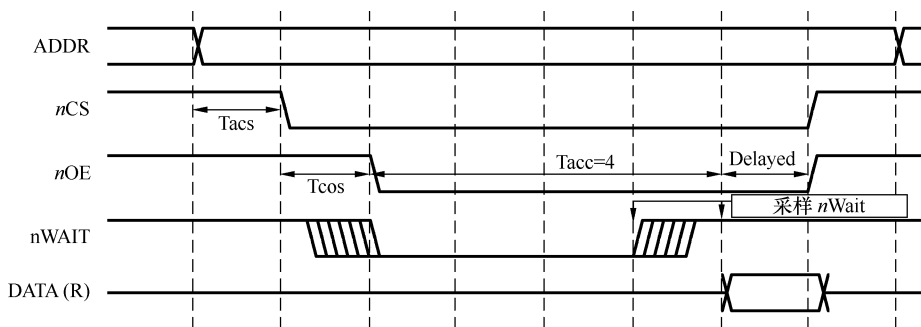


图 2.27 SRAM 读时序图

NOR Flash 和许多外设控制芯片都使用了类似 SRAM 的访问时序, 因此, 牢固掌握这个时序意义重大。一般, 在芯片数据手册给出的时序图中, 会给出图中各段时间的含义和要求, 真实的电路板必须满足芯片手册上描述的建立时间和保持时间的最小要求。

2.7 芯片手册阅读方法

芯片手册往往长达数百页甚至上千页，而且全部是英文，从头到尾不加区分地阅读需要花费非常长的时间，而且不一定能获取对设计设备驱动有帮助的信息。芯片手册的正确阅读方法是快速而准确地定位有用信息，重点阅读这些信息，忽略无关内容。下面以 S3C6410A 的 datasheet 为例来分析阅读方法，为了直观地反映阅读过程，本节的图都直接从手册中抓屏而得。

打开 S3C6410 A 的 datasheet，发现页数为 1378 页，阅读这样的数据手册所花费的时间足够完成整个驱动的设计工作了。

S3C6410A datasheet 的第 1 章“PRODUCT OVERVIEW”即“产品综述”是必读的，通过阅读这一部分可以获知整个芯片的组成。这一章往往会给出一个芯片的整体结构图，并对芯片内的主要模块进行一个简洁的描述。如图 2.28 所示，S3C6410A 的整体结构图在第 61 页出现。

1.2 FEATURES

This section summarizes the features of the S3C6410X. Figure 1-1 is an overall block diagram of the S3C6410X.

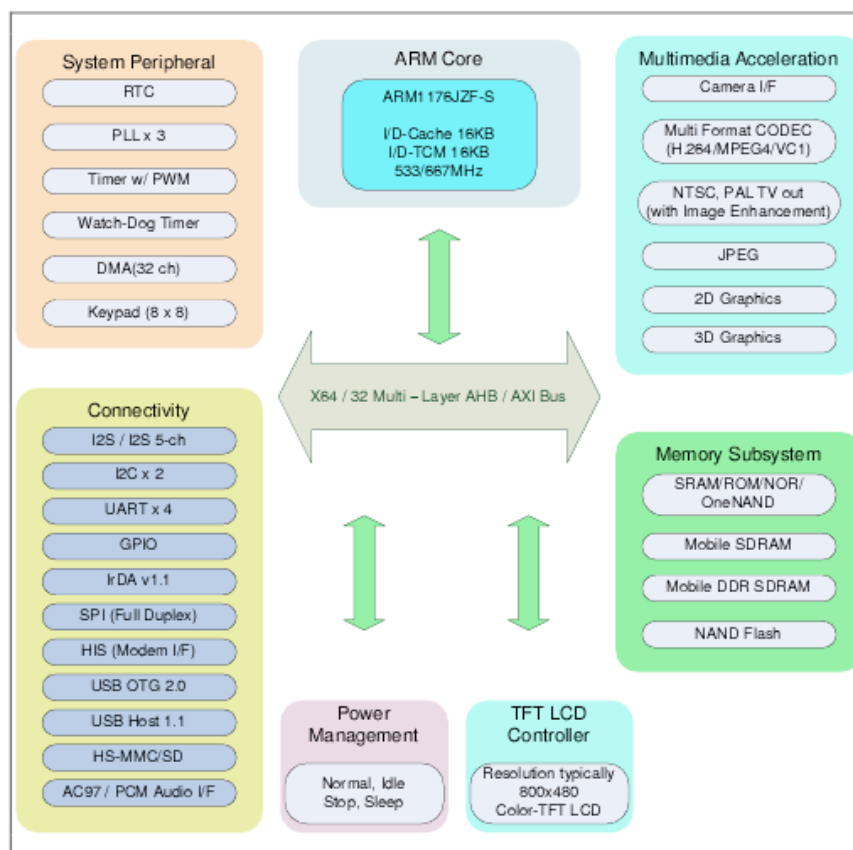


图 2.28 S3C6410A datasheet 中芯片结构图



第 2~43 章每一章都对应 S3C6410A 整体结构图中的一个模块, 图 2.29 为从 Adobe Acrobat 中直接抓出的 S3C6410A datasheet 目录结构图。

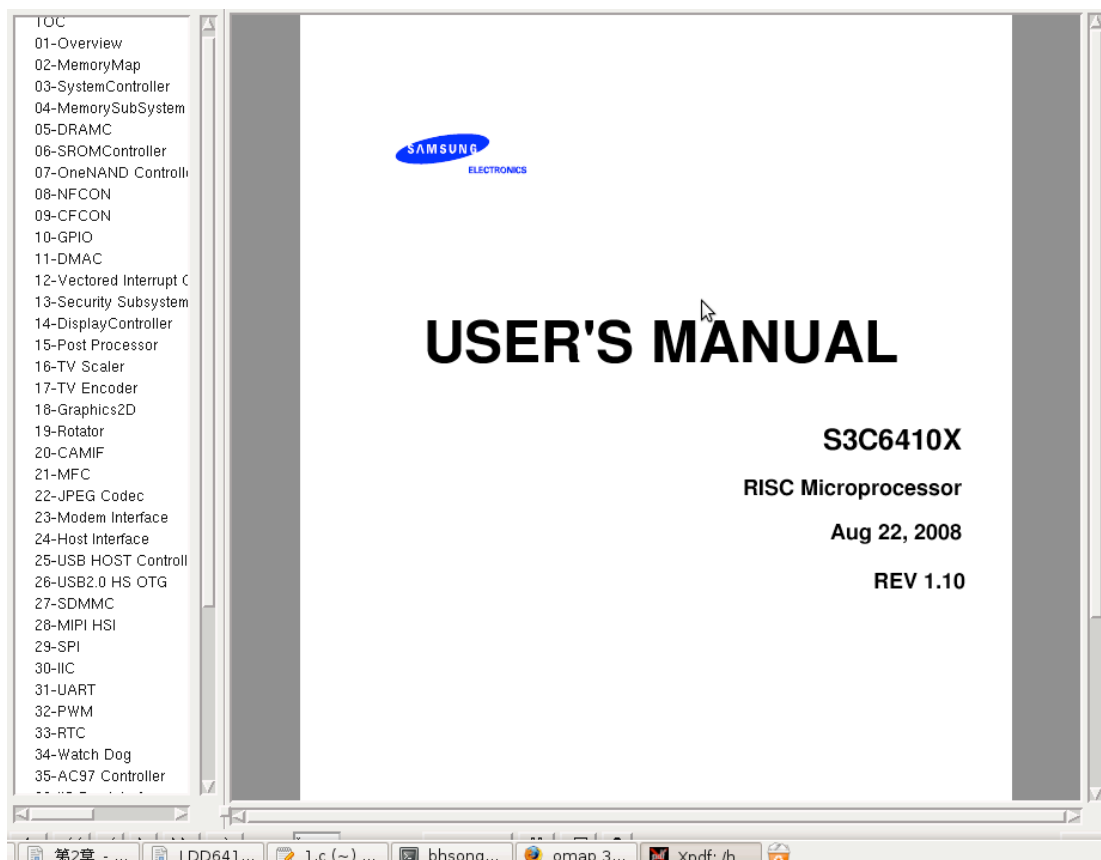


图 2.29 S3C6410A datasheet 目录结构

第 2 章“MemoryMap”即“内存映射”比较关键, 对于定位存储器和外设所对应的基地址有直接指导意义, 这一部分应该细看。

第 3~34 章对应于 CPU 内部集成的外设或总线控制器, 当具体编写某一接口的驱动时, 应该详细阅读, 主要是要分析数据、控制、地址寄存器 (datasheet 中一般会以表格列出) 的访问控制和具体设备的操作流程 (datasheet 中会给出步骤, 有的还会给出流程图)。譬如为了编写 S3C6410A 的 I²C 控制器驱动, 我们需要详细阅读类似图 2.30 的寄存器定义表格和图 2.31 的操作流程图。

第 44 章“ELECTRICAL DATA”即“电气数据”, 描述芯片的电气特性, 如电压、电流和各种工作模式下的时序及建立时间和保持时间的要求。所有的 datasheet 都会包含类似章节, 这一章对于硬件工程师比较关键, 但是, 一般来说, 驱动工程师并不需阅读。

第 45 章“MECHANICAL DATA”即“机械数据”, 描述芯片的物理特性、尺寸和封装, 硬件工程师会依据这一章绘制芯片的封装 (footprint), 但是, 驱动工程师无需阅读。

30.11.1 MULTI-MASTER IIC-BUS CONTROL (IICCON) REGISTER

Register	Address	R/W	Description	Reset Value
IICCON	0x7F004000	R/W	IIC Channel 0 Bus control register	0x0X
	0x7F00F000	R/W	IIC Channel 1 Bus control register	0x0X

IICCON	Bit	Description	Initial State
Acknowledge generation (1)	[7]	IIC-bus acknowledge (ACK) enable bit. 0: Disable 1: Enable In Tx mode, the IICSDA is free in the ACK time. In Rx mode, the IICSDA is L in the ACK time.	0
Tx clock source selection	[6]	Source clock of IIC-bus transmit clock prescaler selection bit. 0: IICCLK = PCLK /16 1: IICCLK = PCLK /512	0
Tx/Rx Interrupt (5)	[5]	IIC-Bus Tx/Rx interrupt enable/disable bit. 0: Disable, 1: Enable	0
Interrupt pending flag (2) (3)	[4]	IIC-bus Tx/Rx interrupt pending flag. This bit cannot be written to 1. When this bit is read as 1, the IIC_SCL is tied to L and the IIC is stopped. To resume the operation, clear this bit as 0. 0: 1) No interrupt pending (when read). 2) Clear pending condition & Resume the operation (when write). 1: 1) Interrupt is pending (when read) 2) N/A (when write)	0
Transmit clock value (4)	[3:0]	IIC-Bus transmit clock prescaler. IIC-Bus transmit clock frequency is determined by this 4-bit prescaler value, according to the following formula: Tx clock = IICCLK/(IICCON[3:0]+1).	Undefined

图 2.30 芯片手册中以表格形式列出寄存器定义

30.10 FLOWCHARTS OF OPERATIONS IN EACH MODE

The following steps must be executed before any IIC Tx/Rx operations.

1. Write own slave address on IICADD register, if needed.
2. Set IICCON register.
 - a) Enable interrupts
 - b) Define SCL period
3. Set IICSTAT to enable Serial Output

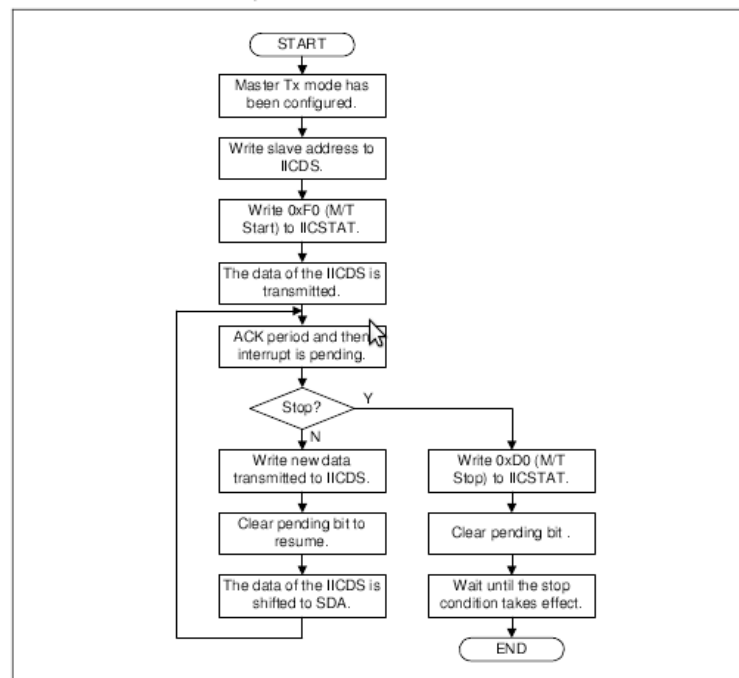


图 2.31 芯片手册中给出外设控制器的操作流程



2.8 仪器仪表使用

2.8.1 万用表

在电路板调试过程中我们主要使用万用表的两个功能。

- 测量电平。
- 使用二极管挡测量电路板上网络的连通性，当示波器被设置在二极管挡，测量连通的网路会发出“滴滴”的鸣叫，否则，没有连通。

2.8.2 示波器

示波器是利用电子示波管的特性，将人眼无法直接观测的交变电信号转换成图像，显示在荧光屏上以便测量的电子仪器。它是观察数字电路实验现象、分析实验中的问题、测量实验结果必不可少的重要仪器。

使用示波器主要应注意调节垂直偏转因数选择 (VOLTS/DIV) 和微调、时基选择 (TIME/DIV) 和微调以及触发方式。

如果 VOLTS/DIV 设置不合理，则可能造成电压幅度超出整个屏幕或在屏幕上变动太过微小无法观测的现象。图 2.32 所示为同一个波形在 VOLTS/DIV 设置由大到小变化过程中的示意图。

如果 TIME/DIV 设置不合适，则可能造成波形混迭。混迭意味着屏幕上显示的波形频率低于信号的实际频率。这时候，可以通过慢慢改变扫速 TIME/DIV 到较快的时基挡，如果波形的频率参数急剧改变或者晃动的波形在某个较快的时基挡稳定下来，说明之前发生了波形混迭。根据奈奎斯特定理，采样速率至少高于信号高频成分的 2 倍才不会发生混迭，如一个 500MHz 的信号，

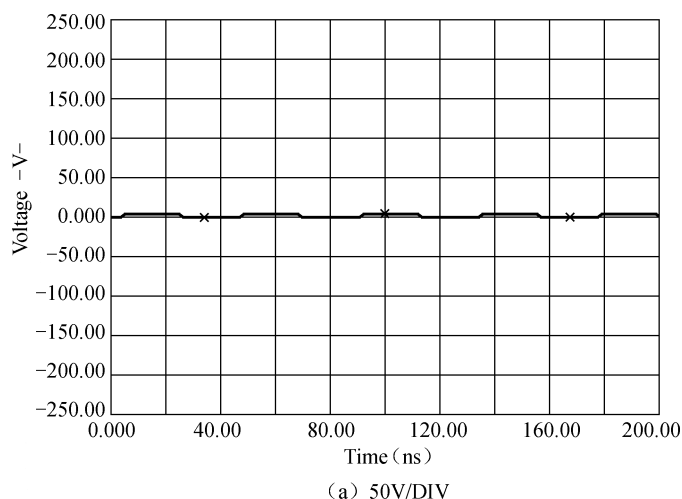


图 2.32 示波器的 VOLTS/DIV 设置与波形

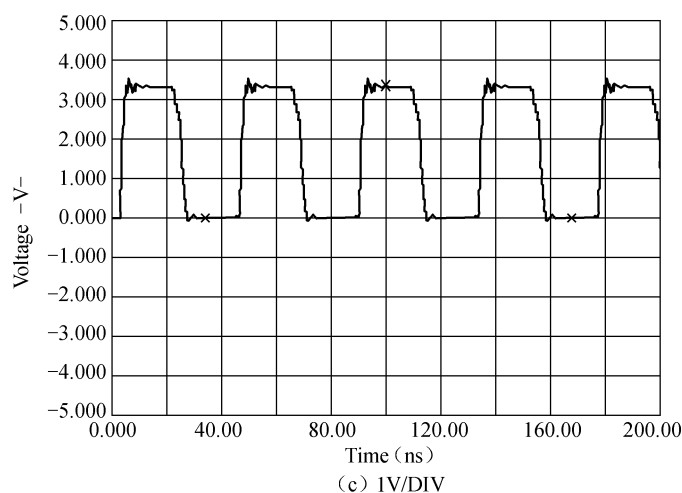
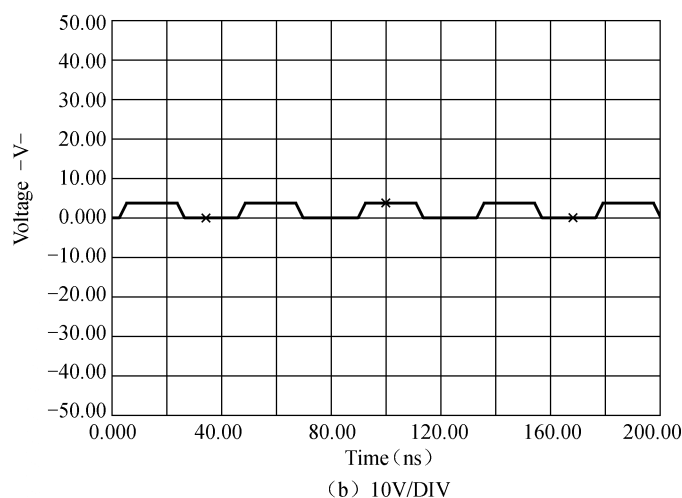


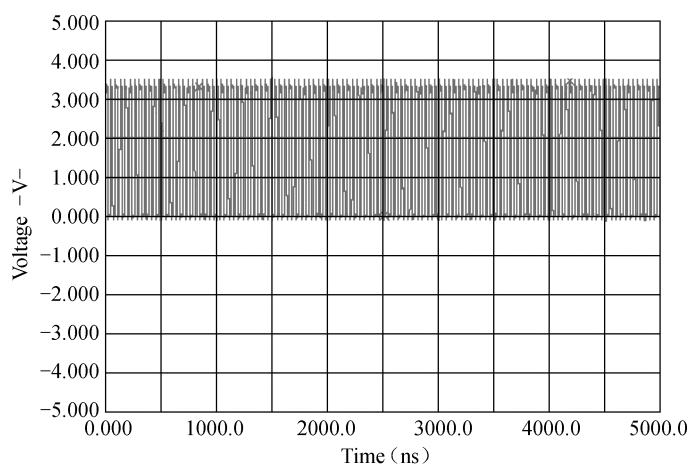
图 2.32 示波器的 VOLTS/DIV 设置与波形 (续)

至少需要 1GS/s 的采样速率。图 2.33 所示为同一个波形在 TIME/DIV 设置由小到大变化过程中的示意图。

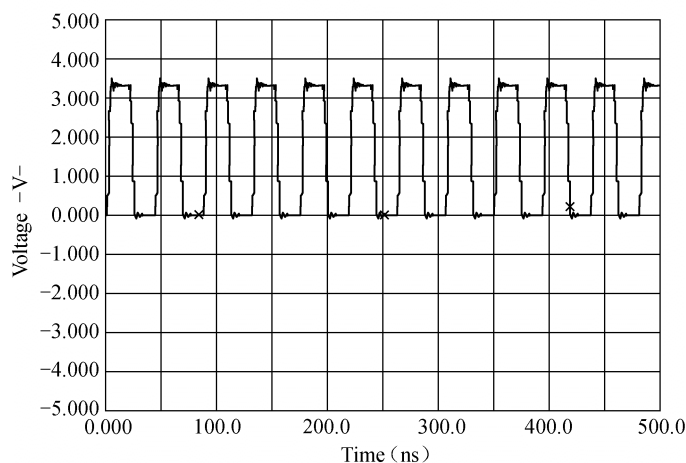


奈奎斯特定理即为采样定理，指当采样频率 $f_{s_{\max}}$ 大于信号中最高频率 f_{\max} 的 2 倍时，即 $f_{s_{\max}} \geq 2f_{\max}$ 时，采样之后的数字信号可完整地保留了原始信息。这条定理在信号处理领域的地位相当之高，大致相当于物理学领域的牛顿定律。

示波器的触发能使信号在正确的位置点同步水平扫描，使信号特性清晰。触发控制按钮可以稳定重复的波形并捕获单次波形。大多数用示波器的用户只采用边沿触发方式，如果拥有其他触发能力在某些应用上是非常有用的，特别是对新设计产品的故障查询，先进的触发方式可将所关心的事件分离出来，找出用户关心的非正常问题，从而最有效地利用采样速率和存储深度。触发能力的提高可以较大提高测试过程的灵活性。



(a) 500ns/DIV



(b) 50ns/DIV

图 2.33 示波器的 TIME/DIV 设置与波形

2.8.3 逻辑分析仪

逻辑分析仪是利用时钟从测试设备上采集数字信号并进行显示的仪器，其最主要的作用是用于时序的判定。与示波器不同，逻辑分析仪并不具备许多电压等级，通常只显示两个电压（逻辑 1 和 0）。在设定了参考电压之后，逻辑分析仪对待测试信号通过比较器来进行判定，高于参考电压者为 High，低于参考电压者为 Low。

例如，如果以 n MHz 采样率测量一个信号，逻辑分析仪会以 $1\,000/n$ ns 为周期采样信号，当参考电压设定为 1.5V 时，超过 1.5V 则判定为 1，低于 1.5V 则为 0，将逻辑 1 和 0 连接成连续的波形，工程师依据此连续波形可寻找时序问题。

高端的逻辑分析仪会安装有 Windows XP 操作系统并提供非常友善的逻辑分析应用软件，在其中可方便地编辑探针、信号并察看波形，如图 2.34 所示。

逻辑分析仪的波形可以显示地址、数据、控制信号及任意外部探针信号的变化轨迹，在使用之前应先编辑每个探针的信号名。

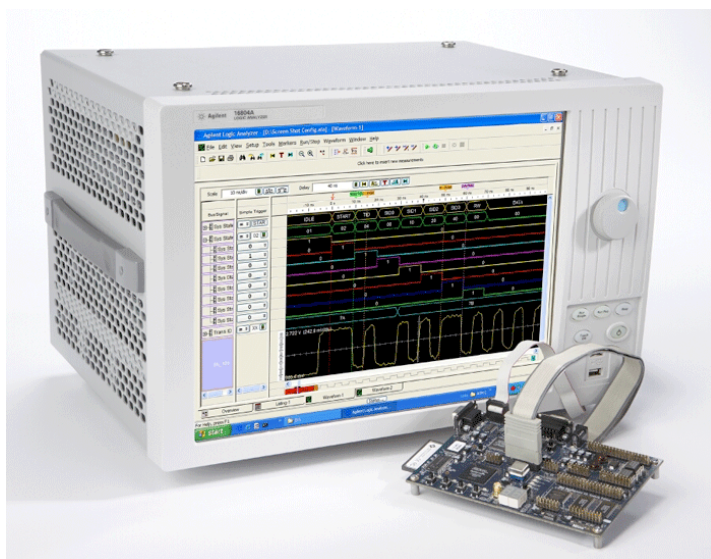


图 2.34 逻辑分析仪及配套软件

逻辑分析仪具有超强的逻辑跟踪分析功能，它可以捕获并记录嵌入式处理器的总线周期，也可以捕获如实时跟踪用的 ETM 接口的程序执行信息，并对这些记录进行分析、译码并还原出应用程序的执行过程。因此，可使用逻辑分析仪通过触发接口与 ICD（在线调试器）协调工作以补充 ICD 在跟踪功能方面的不足。逻辑分析仪与 ICD 协作可为工程师提供断点、触发和跟踪调试手段，如图 2.35 所示。



ICD 是一个容易与 ICE（在线仿真器）混淆的概念，ICE 本身需要完全仿真 CPU 的行为，可以从物理上完全替代 CPU，而 ICD 则只是与芯片内部提供的嵌入式 ICE 单元通过 JTAG 等接口互通。因此，对 ICD 的硬件性能要求远低于 ICE。目前市面上出现的很多号称 ICE 的产品实际上是 ICD，如 ARM 公司的 Multi-ICE、WindRiver 公司的 visionICE 和 visionProbe 等。

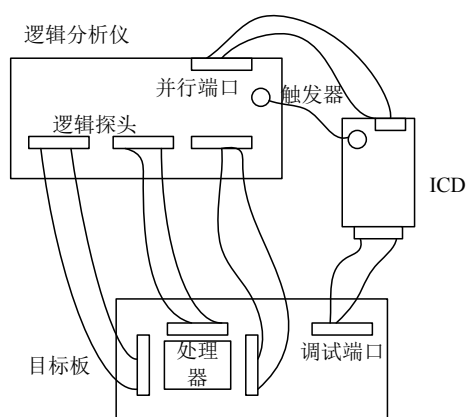


图 2.35 逻辑分析仪与 ICD 协作



2.9 总结

本章简单地讲解了驱动软件工程师必备的硬件基础知识，描述了处理器、存储器的分类以及各种处理器、存储器的原理与用途，并分析了常见的外围设备接口与总线的工作方式。

此外，本章还讲述了对驱动工程师进行实际项目开发有帮助作用的原理图、硬件时序分析方法，数据手册阅读方法以及万用表、示波器和逻辑分析仪的使用方法。

LINUX

第3章

Linux 内核及内核编程

本章导读

本章为读者打下 Linux 驱动编程的软件基础。由于 Linux 驱动编程本质属于 Linux 内核编程，因此我们有必要熟悉 Linux 内核及内核编程的基础知识。

3.1~3.2 节讲解了 Linux 内核的演变及新版 Linux 2.6 内核的特点。

3.3 节分析了 Linux 内核源代码目录结构和 Linux 内核的组成部分及其关系，并对 Linux 的用户空间和内核空间进行了说明。

3.4 节讲述了 Linux 2.6 内核的编译及内核引导过程。除此之外，还描述了在 Linux 内核中新增程序的方法，驱动工程师编写的设备驱动也应该以此方式被添加。

3.5 节阐述了 Linux 下 C 编程的命名习惯以及 Linux 所使用的 GNU C 针对标准 C 的扩展语法。





3.1 Linux 内核的发展与演变

Linux 操作系统是 UNIX 操作系统的一种克隆系统, 诞生于 1991 年 10 月 5 日 (第一次正式向外公布的时间)。Linux 操作系统的诞生、发展和成长过程依赖着 5 个重要支柱: UNIX 操作系统、Minix 操作系统、GNU 计划、Posix 标准和 Internet。

1. UNIX 操作系统

UNIX 操作系统是美国贝尔实验室的 Ken. Thompson 和 Dennis Ritchie 于 1969 年夏在 DEC PDP-7 小型计算机上开发的一个分时操作系统。Linux 操作系统可看作 UNIX 操作系统的一个克隆版本。

2. Minix 操作系统

Minix 操作系统也是 UNIX 的一种克隆系统, 它于 1987 年由著名计算机教授 Andrew S. Tanenbaum 开发完成。开放源代码 Minix 系统的出现在全世界的大学中刮起了学习 UNIX 系统的旋风。Linux 刚开始就是参照 Minix 系统于 1991 年才开始开发。

3. GNU 计划

GNU 计划和自由软件基金会 (FSF) 是由 Richard M. Stallman 于 1984 年创办的, GNU 是 “GNU's Not UNIX” 的缩写。到 20 世纪 90 年代初, GNU 项目已经开发出许多高质量的免费软件, 其中包括 emacs 编辑系统、bash shell 程序、gcc 系列编译程序、gdb 调试程序等。这些软件为 Linux 操作系统的开发创造了一个合适的环境, 是 Linux 诞生的基础之一。没有 GNU 软件环境, Linux 将寸步难行。因此, 严格而言, “Linux” 应该被称为 “GNU/Linux” 系统。

4. Posix 标准

Posix (Portable Operating System Interface for Computing Systems, 可移植的操作系统接口) 是由 IEEE 和 ISO/IEC 开发的一组标准。该标准基于现有的 UNIX 实践和经验完成, 描述了操作系统的调用服务接口, 用于保证编制的应用程序可以在源代码一级上在多种操作系统上移植。该标准在推动 Linux 操作系统朝着正规化发展起着重要的作用, 是 Linux 前进的灯塔。

5. Internet

如果没有 Internet, 没有遍布全世界的无数计算机骇客的无私奉献, 那么 Linux 最多只能发展到 0.13 (0.95) 版的水平。从 0.95 版开始, 对内核的许多改进和扩充均以其他人为主了, 而 Linus 以及其他 maintainer 的主要任务开始变成对内核的维护和决定是否采用某个补丁程序。

表 3.1 描述了 Linux 操作系统重要版本的变迁历史及各版本的主要特点。

表 3.1 Linux 操作系统版本历史

版 本	时 间	特 点
0.1	1991.10	最初的原型
1.0	1994.3	包含了 386 的官方支持, 仅支持单 CPU 系统
1.2	1995.3	第一个包含多平台 (Alpha、Sparc、MIPS 等) 支持的官方版本
2.0	1996.6	包含很多新的平台支持, 最重要的是, 它是第一个支持 SMP (对称多处理器) 体系的内核版本

续表

版 本	时 间	特 点
2.2	1999.1	极大提升 SMP 系统上 Linux 的性能，并支持更多的硬件
2.4	2001.1	进一步地提升了 SMP 系统的扩展性，同时也集成了很多用于支持桌面系统的特性：USB、PC 卡（PCMCIA）的支持，内置的即插即用等
2.6	2003.12	无论是对于企业服务器还是对于嵌入式系统，Linux 2.6 都是一个巨大的进步。对高端的机器来说，新特性针对的是性能改进、可扩展性、吞吐率，以及对 SMP 机器 NUMA 的支持。对于嵌入式领域，添加了新的体系结构和处理器类型。包括对那些没有硬件控制的内存管理方案的 MMU-less 系统的支持。同样地，为了满足桌面用户群的需要，添加了一整套新的音频和多媒体驱动程序

从表 3.1 可以看出，Linux 的开发一直朝着支持更多的 CPU、硬件体系结构和外部设备，支持更广泛领域的应用，提供更好的性能 3 个方向发展。

除了 Linux 内核本身可提供免费下载以外，一些厂商封装了 Linux 内核和大量有用的软件包，制定了相应的 Linux 发布版，如 Red Hat Linux、TurboLinux、Debian、SuSe、Ubuntu，国内的 RedFlag 和 xteam 等。

再者，针对嵌入式系统的应用，一些改进内核的 Linux 被开发出来，如改进实时性的 Hard Hat Linux 和 RTLinux、支持不含 MMU CPU 的 μ Clinux（日前 Linux mainline 已经支持 MMU-less 系统）、面向数字相机和 MP3 等微型嵌入式设备的 ThinLinux 和以及颇有商业背景的 MontaVista 等。

3.2 Linux 2.6 内核的特点

本书基于的是 Linux 2.6 内核，LDD6410 开发板内核的完整版本号为 2.6.28.6。Linux 2.6 内核是 Linux 开发者群落一个寄予厚望的版本，从 2003 年 12 月 Linux 2.6.0 发布至今，一直还处于开发之中，并还将稳定较长一段时间。Linux 2.6 相对于 Linux 2.4 有相当大的改进，主要体现在如下几个方面：

1. 新的调度器

2.6 版本的 Linux 内核使用了新的进程调度算法，它在高负载的情况下执行得极其出色，并且当有很多处理器时也可以很好地扩展。

2. 内核抢占

在 2.6 版本的 Linux 内核中，一个内核任务可以被抢占，从而提高系统的实时性。这样做最主要的优势在于，可以极大地增强系统的用户交互性，用户将会觉得鼠标单击和击键的事件得到了更快速的响应。

3. 改进的线程模型

2.6 版本的 Linux 中线程操作速度得以提高，可以处理任意数目的线程，最大可以到 20 亿。

4. 虚拟内存的变化

从虚拟内存的角度来看，新内核融合了 r-map（反向映射）技术，显著改善虚拟内存存在一定程度负载下的性能。



5. 文件系统

2.6 版内核增加了对日志文件系统功能的支持, 解决了 2.4 版在这方面的不足。2.6 版内核在文件系统上的关键变化还包括对扩展属性及 Posix 标准访问控制的支持。ext2/ext3 作为大多数 Linux 系统缺省安装的文件系统, 在 2.6 版内核中增加了对扩展属性的支持, 可以给指定的文件在文件系统中嵌入元数据。

6. 音频

新的 Linux 音频体系结构 ALSA (Advanced Linux Sound Architecture) 取代了缺陷很多的旧的 OSS (Open Sound System)。新的声音体系结构支持 USB 音频和 MIDI 设备, 并支持全双工重放等功能。

7. 总线

SCSI/IDE 子系统经过大幅度的重写, 解决和改善了以前的一些问题。比如 2.6 版内核可以直接通过 IDE 驱动程序来支持 IDE CD/RW 设备, 而不必像以前一样要使用一个特别的 SCSI 模拟驱动程序。

8. 电源管理

支持 ACPI (高级电源配置管理界面, Advanced Configuration and Power Interface), 用于调整 CPU 在不同的负载下工作于不同的时钟频率以降低功耗。

9. 联网和 IPSec

2.6 内核中加入了 IPSec 的支持, 删除了原来内核内置的 HTTP 服务器 khttpd, 加入了对新的 NFSv4 (网络文件系统) 客户机/服务器的支持, 并改进了对 IPv6 的支持。

10. 用户界面层

2.6 内核重写了帧缓冲/控制台层, 人机界面层还加入了对近乎所有接口设备的支持 (从触摸屏到盲人用的设备和各种各样的鼠标)。

在设备驱动程序的方面, Linux 2.6 相对于 Linux 2.4 也有较大的改动, 这主要表现在内核 API 中增加了不少新功能 (例如内存池)、sysfs 文件系统、内核模块从.o 变为.ko、驱动模块编译方式、模块使用计数、模块加载和卸载函数的定义等方面。

3.3 Linux 内核的组成

3.3.1 Linux 内核源代码目录结构

本书范例程序所基于的 Linux 2.6.28.6 内核源代码包含如下目录。

- arch: 包含和硬件体系结构相关的代码, 每种平台占一个相应的目录, 如 i386、arm、powerpc、mips 等。
- block: 块设备驱动程序 I/O 调度。
- crypto: 常用加密和散列算法 (如 AES、SHA 等), 还有一些压缩和 CRC 校验算法。
- Documentation: 内核各部分的通用解释和注释。
- drivers: 设备驱动程序, 每个不同的驱动占用一个子目录, 如 char、block、net、mtd、i2c 等。

- fs: 支持的各种文件系统, 如 EXT、FAT、NTFS、JFFS2 等。
- include: 头文件, 与系统相关的头文件被放置在 include/linux 子目录下。
- init: 内核初始化代码。
- ipc: 进程间通信的代码。
- kernel: 内核的最核心部分, 包括进程调度、定时器等, 而和平台相关的一部分代码放在 arch/*/kernel 目录下。
- lib: 库文件代码。
- mm: 内存管理代码, 和平台相关的一部分代码放在 arch/*/mm 目录下。
- net: 网络相关代码, 实现了各种常见的网络协议。
- scripts: 用于配置内核的脚本文件。
- security: 主要是一个 SELinux 的模块。
- sound: ALSA、OSS 音频设备的驱动核心代码和常用设备驱动。
- usr: 实现了用于打包和压缩的 cpio 等。

3.3.2 Linux 内核的组成部分

如图 3.1 所示, Linux 内核主要由进程调度 (SCHED)、内存管理 (MM)、虚拟文件系统 (VFS)、网络接口 (NET) 和进程间通信 (IPC) 5 个子系统组成。

1. 进程调度

进程调度控制系统中的多个进程对 CPU 的访问, 使得多个进程能在 CPU 中“微观串行, 宏观并行”地执行。进程调度处于系统的中心位置, 内核中其他的子系统都依赖它, 因为每个子系统都需要挂起或恢复进程。

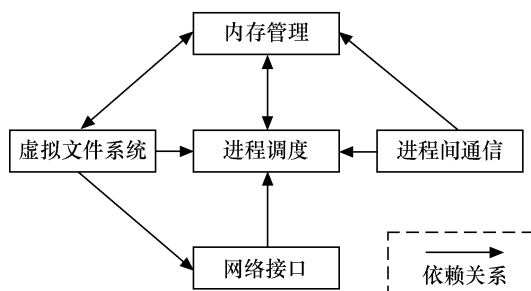


图 3.1 Linux 内核的组成部分与关系

如图 3.2 所示, Linux 的进程在几个状态间进行切换。在设备驱动编程中, 当请求的资源不能得到满足时, 驱动一般会调度其他进程执行, 并使本进程进入睡眠状态, 直到它请求的资源被释放, 才会被唤醒而进入就绪态。睡眠分成可被打断的睡眠和不可被打断的睡眠, 两者的区别在于可被打断的睡眠在收到信号的时候会醒。

在设备驱动编程中, 当请求的资源不能得到满足时, 驱动一般会调度其他进程执行, 其对应进程进入睡眠状态, 直到它请求的资源被释放, 才会被唤醒而进入就绪态。

设备驱动中, 如果需要几个并发执行的任务, 可以启动内核线程, 启动内核线程的函数为:

```
pid_t kernel_thread(int (*fn)(void *), void *arg, unsigned long flags);
```

2. 内存管理

内存管理的主要作用是控制多个进程安全地共享主内存区域。当 CPU 提供内存管理单元 (MMU) 时, Linux 内存管理完成为每个进程进行虚拟内存到物理内存的转换。Linux 2.6 引入了对无 MMU CPU 的支持。

如图 3.3 所示, 一般而言, Linux 的每个进程享有 4GB 的内存空间, 0~3GB 属于用户空间, 3~4GB 属于内核空间, 内核空间对常规内存、I/O 设备内存以及高端内存存在不同的处理方式。

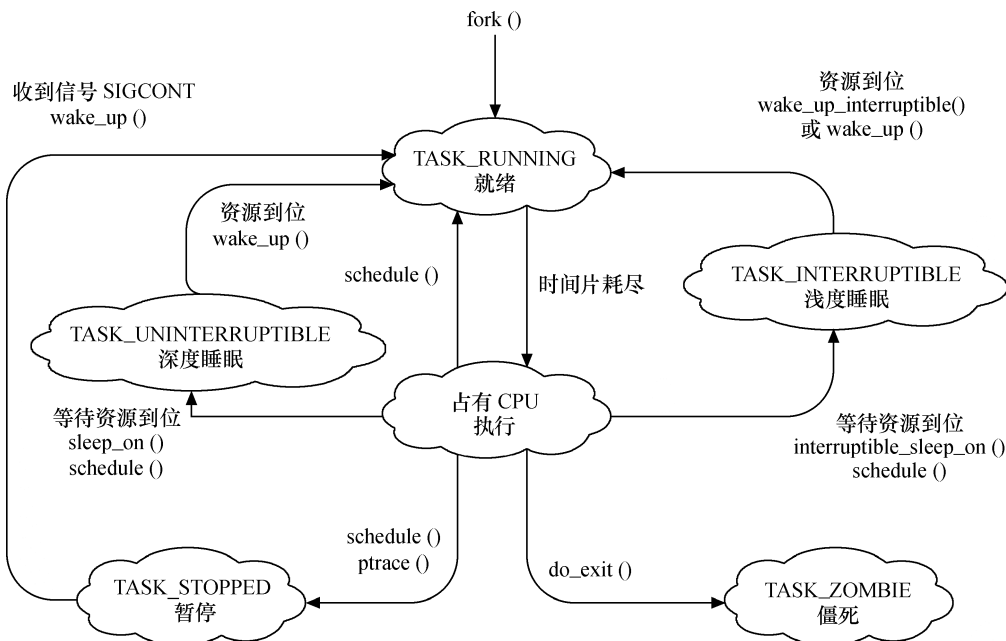


图 3.2 Linux 进程状态转换

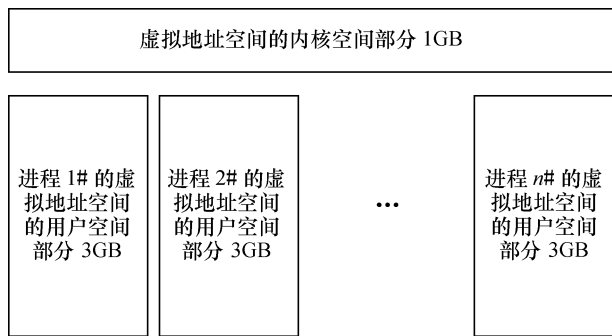


图 3.3 Linux 进程地址空间

3. 虚拟文件系统

如图 3.4 所示, Linux 虚拟文件系统 (VFS) 隐藏各种了硬件的具体细节, 为所有的设备提供了统一的接口。而且, 它独立于各个具体的文件系统, 是对各种文件系统的一个抽象, 它使用超

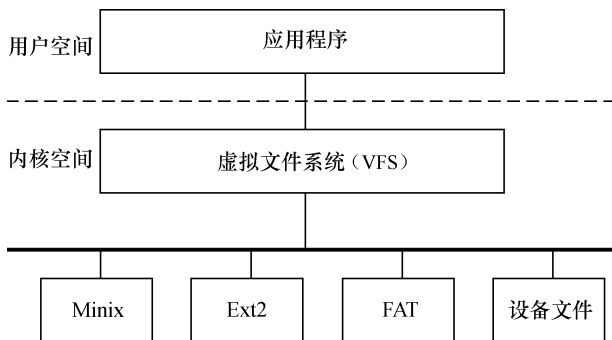


图 3.4 Linux 文件系统

级块 super block 存放文件系统相关信息，使用索引节点 inode 存放文件的物理信息，使用目录项 dentry 存放文件的逻辑信息。

4. 网络接口

网络接口提供了对各种网络标准的存取和各种网络硬件的支持。如图 3.5 所示，在 Linux 中网络接口可分为网络协议和网络驱动程序，网络协议部分负责实现每一种可能的网络传输协议，网络设备驱动程序负责与硬件设备通信，每一种可能的硬件设备都有相应的设备驱动程序。

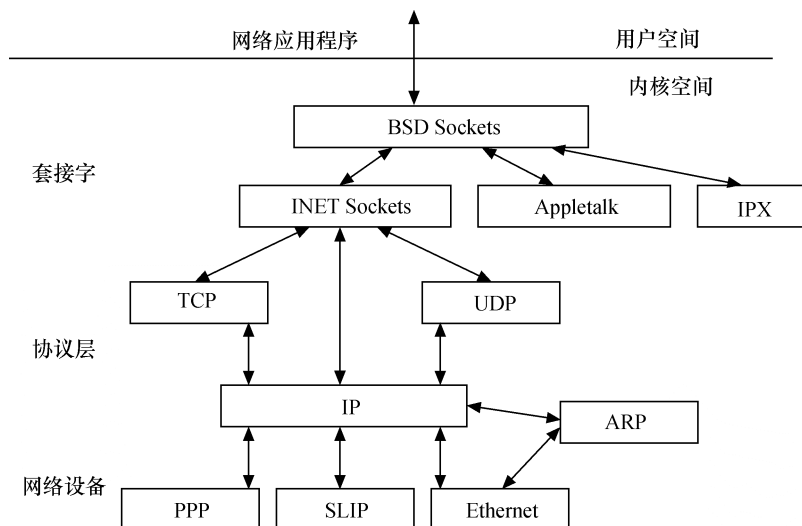


图 3.5 Linux 网络体系结构

5. 进程通信

进程通信支持提供进程之间的通信，Linux 支持进程间的多种通信机制，包含信号量、共享内存、管道等，这些机制可协助多个进程、多资源的互斥访问、进程间的同步和消息传递。

Linux 内核的 5 个组成部分之间的依赖关系如下。

- 进程调度与内存管理之间的关系：这两个子系统互相依赖。在多道程序环境下，程序要运行必须为之创建进程，而创建进程的第一件事情，就是将程序和数据装入内存。
- 进程间通信与内存管理的关系：进程间通信子系统要依赖内存管理支持共享内存通信机制，这种机制允许两个进程除了拥有自己的私有空间，还可以存取共同的内存区域。
- 虚拟文件系统与网络接口之间的关系：虚拟文件系统利用网络接口支持网络文件系统 (NFS)，也利用内存管理支持 RAMDISK 设备。
- 内存管理与虚拟文件系统之间的关系：内存管理利用虚拟文件系统支持交换，交换进程 (swpd) 定期由调度程序调度，这也是内存管理依赖于进程调度的惟一原因。当一个进程存取的内存映射被换出时，内存管理向文件系统发出请求，同时，挂起当前正在运行的进程。

除了这些依赖关系外，内核中的所有子系统还要依赖于一些共同的资源。这些资源包括所有子系统都用到的例程，如分配和释放内存空间的函数、打印警告或错误信息的函数及系统提供的调试例程等。



3.3.3 Linux 内核空间与用户空间

现代 CPU 内部往往实现了不同的操作模式（级别），不同的模式有不同的功能，高层程序往往不能访问低级功能，而必须以某种方式切换到低级模式。

例如，ARM 处理器分为 7 种工作模式。

- 用户模式 (usr)：大多数的应用程序运行在用户模式下，当处理器运行在用户模式下时，某些被保护的系统资源是不能被访问的。
- 快速中断模式 (fiq)：用于高速数据传输或通道处理。
- 外部中断模式 (irq)：用于通用的中断处理。
- 管理模式 (svc)：操作系统使用的保护模式。
- 数据访问终止模式 (abt)：当数据或指令预取终止时进入该模式，可用于虚拟存储及存储保护。
- 系统模式 (sys)：运行具有特权的操作系统任务。
- 未定义指令中止模式 (und)：当未定义的指令执行时进入该模式，可用于支持硬件协处理器的软件仿真。

ARM Linux 的系统调用实现原理是采用 swi 软中断从用户态 usr 模式陷入内核态 svc 模式。

又如，X86 处理器包含 4 个不同的特权级，称为 Ring 0~Ring 3。Ring 0 下，可以执行特权级指令，对任何 I/O 设备都有访问权等，而 Ring 3 则被限制很多操作。

Linux 系统充分利用 CPU 的这一硬件特性，但它只使用了两级。在 Linux 系统中，内核可进行任何操作，而应用程序则被禁止对硬件的直接访问和对内存的未授权访问。例如，若使用 X86 处理器，则用户代码运行在特权级 3，而系统内核代码则运行在特权级 0。

内核空间和用户空间这两个名词被用来区分程序执行的这两种不同状态，它们使用不同的地址空间。Linux 只能通过系统调用和硬件中断完成从用户空间到内核空间的控制转移。

3.4 Linux 内核的编译及加载

3.4.1 Linux 内核的编译

Linux 驱动工程师需要牢固地掌握 Linux 内核的编译方法以为嵌入式系统构建可运行的 Linux 操作系统映像。在编译 LDD6410 的内核时，需要配置内核，可以使用下面命令中的一个：

```
#make config (基于文本的最为传统的配置界面，不推荐使用)
#make menuconfig (基于文本菜单的配置界面)
#make xconfig (要求 QT 被安装)
#make gconfig (要求 GTK+被安装)
```

在配置 Linux 2.6 内核所使用的 make config、make menuconfig、make xconfig 和 make gconfig 这 4 种方式中，最值得推荐的是 make menuconfig，它不依赖于 QT 或 GTK+，且非常直观，对 LDD6410 的 Linux 2.6.28 内核运行 make menuconfig 后的界面如图 3.6。

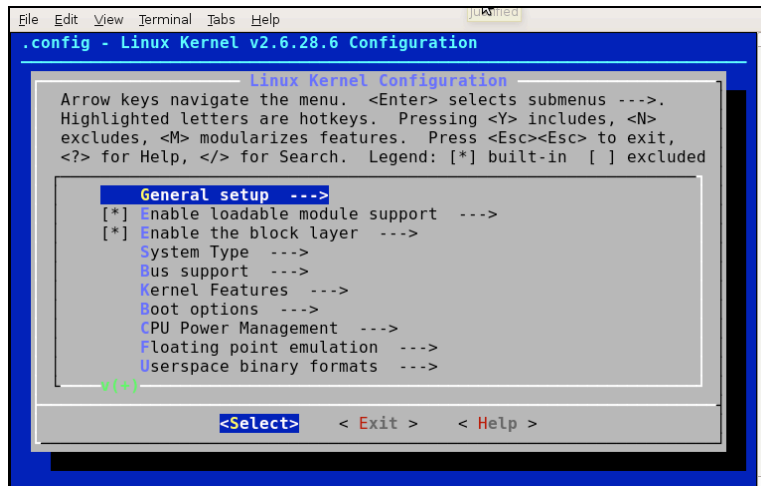


图 3.6 Linux 内核编译配置

内核配置包含的项目相当多，arch/arm/configs/ldd6410lcd_defconfig 文件包含了 LDD6410 的默认配置，因此，只需要运行 make ldd6410lcd_defconfig 就可以为 LDD6410 开发板配置内核。

编译内核和模块的方法是：

```
make zImage
make modules
```

执行完上述命令后，在源代码的根目录下会得到未压缩的内核映像 vmlinux 和内核符号表文件 System.map，在 arch/arm/boot/ 目录会得到压缩的内核映像 zImage，在内核各对应目录得到选中的内核模块。

Linux 2.6 内核的配置系统由以下 3 个部分组成。

- Makefile：分布在 Linux 内核源代码中的 Makefile，定义 Linux 内核的编译规则。
- 配置文件（Kconfig）：给用户提供了配置选择的功能。
- 配置工具：包括配置命令解释器（对配置脚本中使用的配置命令进行解释）和配置用户界面（提供基于字符界面和图形界面）。这些配置工具都是使用脚本语言，如 Tcl/Tk、Perl 等编写。

使用 make config、make menuconfig 等命令后，会生成一个 .config 配置文件，记录哪些部分被编译入内核、哪些部分被编译为内核模块。

运行 make menuconfig 等时，配置工具首先分析与体系结构对应的/arch/xxx/Kconfig 文件（xxx 即为传入的 ARCH 参数），/arch/xxx/Kconfig 文件中除本身包含一些与体系结构相关的配置项和配置菜单以外，还通过 source 语句引入了一系列 Kconfig 文件，而这些 Kconfig 又可能再次通过 source 引入下一层的 Kconfig，配置工具依据这些 Kconfig 包含的菜单和项目即可描绘出一个如图 3.6 所示的分层结构。例如，/arch/arm/Kconfig 文件的结构如下：

```
mainmenu "Linux Kernel Configuration"

config ARM
bool
default y
select HAVE_AOUT
```



```
select HAVE_IDE
select RTC_LIB
select SYS_SUPPORTS_APM_EMULATION
select HAVE_OPROFILE
select HAVE_ARCH_KGDB
select HAVE_KPROBES if (!XIP_KERNEL)
select HAVE_KRETPROBES if (HAVE_KPROBES)
select HAVE_FUNCTION_TRACER if (!XIP_KERNEL)
select HAVE_GENERIC_DMA_COHERENT
help
    The ARM series is a line of low-power-consumption RISC chip designs
    licensed by ARM Ltd and targeted at embedded applications and
    handhelds such as the Compaq IPAQ. ARM-based PCs are no longer
    manufactured, but legacy ARM-based PC hardware remains popular in
    Europe. There is an ARM Linux project with a web page at
    <http://www.arm.linux.org.uk/>.

...
config MMU
    bool
    default y

...
config ARCH_S3C64XX
bool "Samsung S3C64XX"
select GENERIC_GPIO
select HAVE_CLK
help
    Samsung S3C64XX series based systems
...
if ARCH_S3C64XX
source "arch/arm/mach-s3c6400/Kconfig"
source "arch/arm/mach-s3c6410/Kconfig"
endif

...
```

3.4.2 Kconfig 和 Makefile

在 Linux 内核中增加程序需要完成以下 3 项工作。

- 将编写的源代码拷入 Linux 内核源代码的相应目录。
- 在目录的 Kconfig 文件中增加关于新源代码对应项目的编译配置选项。
- 在目录的 Makefile 文件中增加对新源代码的编译条目。

1. 实例引导: S3C6410 处理器的 RTC 驱动配置

在讲解 Kconfig 和 Makefile 的语法之前,我们先利用两个简单的实例引导读者建立初步的认识。

首先,在 linux-2.6.28-samsung/drivers rtc 目录中包含了 S3C6410 处理器的 RTC 设备驱动源代码 rtc-s3c.c。

而在该目录的 Kconfig 文件中包含关于 RTC_DRV_S3C 的配置项目:

```
config RTC_DRV_S3C
    tristate "Samsung S3C series SoC RTC"
    depends on ARCH_S3C2410 || ARCH_S3C64XX || ARCH_S5PC1XX || ARCH_S5P64XX
```

```

help
    RTC (Realtime Clock) driver for the clock inbuilt into the
    Samsung S3C24XX series of SoCs. This can provide periodic
    interrupt rates from 1Hz to 64Hz for user programs, and
    wakeup from Alarm.

    The driver currently supports the common features on all the
    S3C24XX range, such as the S3C2410, S3C2412, S3C2413, S3C2440
    and S3C2442.

    This driver can also be build as a module. If so, the module
    will be called rtc-s3c.
  
```

上述 Kconfig 文件的这段脚本意味着只有在 ARCH_S3C2410、ARCH_S3C64XX、ARCH_S5PC1XX 或 ARCH_S5P64XX 项目之一被配置的情况下，才会出现 RTC_DRV_S3C 配置项目，这个配置项目为三态（可编译入内核，可不编译，也可编译为内核模块，选项分别为“Y”、“N”和“M”），菜单上显示的字符串为“Samsung S3C series SoC RTC”，“help”后面的内容为帮助信息。图 3.7 显示了 RTC_DRV_S3C 菜单以及其 help 在运行 make menuconfig 时的情况。

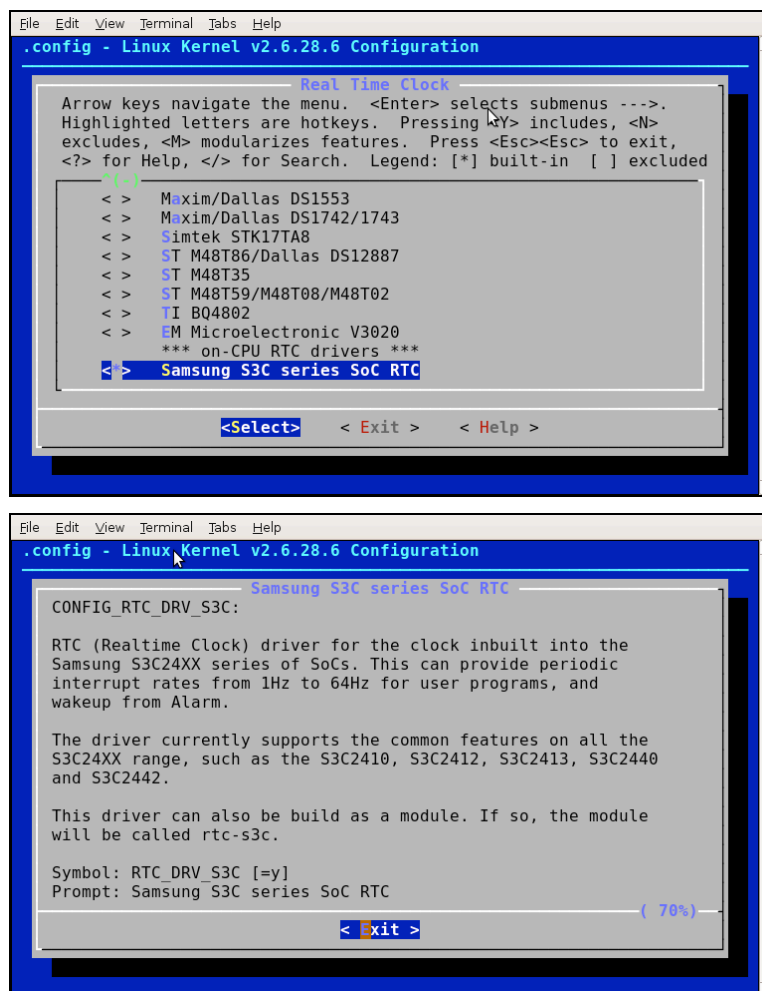


图 3.7 Kconfig 菜单项目与帮助信息



除了布尔型的配置项目外,还存在一种布尔型 (bool) 配置选项,它意味着要么编译入内核,要么不编译,选项为“Y”或“N”。

在目录的 Makefile 中关于 RTC_DRV_S3C 的编译脚本为:

```
obj-$(CONFIG_RTC_DRV_S3C) += rtc-s3c.o
```

上述脚本意味着如果 RTC_DRV_S3C 配置选项被选择为“Y”或“M”,即 obj-\$(CONFIG_RTC_DRV_S3C) 等同于 obj-y 或 obj-m 时,则编译 rtc-s3c.c,选“Y”的情况直接会将生成的目标代码直接连接到内核,为“M”的情况则会生成模块 rtc-s3c.ko;如果 RTC_DRV_S3C 配置选项被选择为“N”,即 obj-\$(CONFIG_RTC_DRV_S3C) 等同于 obj-n 时,则不编译 rtc-s3c.c。

一般而言,驱动工程师只会在内核源代码的 drivers 目录的相应子目录中增加新设备驱动的源代码,并增加或修改 Kconfig 配置脚本和 Makefile 脚本,完全仿照上述过程执行即可。

2. Makefile

这里主要对内核源代码各级子目录中的 kbuild (内核的编译系统) Makefile 进行简单介绍,这部分是内核模块或设备驱动的开发者最常接触到的。

Makefile 的语法包括如下几个方面。

(1) 目标定义。

目标定义就是用来定义哪些内容要作为模块编译,哪些要编译并连接进内核。

例如:

```
obj-y += foo.o
```

表示要由 foo.c 或者 foo.s 文件编译得到 foo.o 并连接进内核,而 obj-m 则表示该文件要作为模块编译。除了 y、m 以外的 obj-x 形式的目标都不会被编译。

而更常见的做法是根据 config 文件的 CONFIG_ 变量来决定文件的编译方式,如:

```
obj-$(CONFIG_ISDN) += isdn.o
obj-$(CONFIG_ISDN_PPP_BSDCOMP) += isdn_bsdcomp.o
```

除了 obj- 形式的目标以外,还有 lib-y library 库,hostprogs-y 主机程序等目标,但是基本都应用在特定的目录和场合下。

(2) 多文件模块的定义。

最简单的 Makefile 如上一节一句话的形式就够了,如果一个模块由多个文件组成,会稍微复杂一些,这时候应采用模块名加-y 或-objs 后缀的形式来定义模块的组成文件。如以下例子:

```
#

# Makefile for the linux ext2-file system routines.
#

obj-$(CONFIG_EXT2_FS) += ext2.o
ext2-y := balloc.o dir.o file.o fsync.o ialloc.o inode.o \
        ioct1.o namei.o super.o symlink.o
ext2-$(CONFIG_EXT2_FS_XATTR) += xattr.o xattr_user.o xattr_trusted.o
ext2-$(CONFIG_EXT2_FS_POSIX_ACL) += acl.o
ext2-$(CONFIG_EXT2_FS_SECURITY) += xattr_security.o
ext2-$(CONFIG_EXT2_FS_XIP) += xip.o
```

模块的名字为 ext2,由 balloc.o、dir.o、file.o 等多个目标文件最终链接生成 ext2.o 直至 ext2.ko 文件,并且是否包括 xattr.o、acl.o 等则取决于内核配置文件的配置情况,例如,如果 CONFIG_EXT2_FS_POSIX_ACL 被选择,则编译 acl.c 得到 acl.o 并最终链接进 ext2。

(3) 目录层次的迭代。

如下例：

```
obj-$(CONFIG_EXT2_FS) += ext2/
```

当 CONFIG_EXT2_FS 的值为 y 或 m 时，kbuild 将会把 ext2 目录列入向下迭代的目标中。

3. Kconfig

内核配置脚本文件的语法也比较简单，主要包括如下几个方面。

(1) 菜单入口。

大多数的内核配置选项都对应 Kconfig 中的一个菜单入口：

```
config MODVERSIONS
    bool "Module versioning support"
    help
        Usually, you have to use modules compiled with your kernel.
        Saying Y here makes it ...
```

“config”关键字定义新的配置选项，之后的几行定义了该配置选项的属性。配置选项的属性包括类型、数据范围、输入提示、依赖关系、选择关系及帮助信息和默认值等。

每个配置选项都必须指定类型，类型包括 bool、tristate、string、hex 和 int，其中 tristate 和 string 是两种基本的类型，其他类型都基于这两种基本类型。类型定义后可以紧跟输入提示，下面的两段脚本是等价的：

```
bool "Networking support"
```

和

```
bool
prompt "Networking support"
```

输入提示的一般格式为：

```
prompt <prompt> [if <expr>]
```

其中可选的 if 用来表示该提示的依赖关系。

默认值的格式为：

```
default <expr> [if <expr>]
```

一个配置选项可以存在任意多个默认值，这种情况下，只有第一个被定义的值是可用的。如果用户不设置对应的选项，配置选项的值就是默认值。

依赖关系的格式为：

```
depends on (或者 requires) <expr>
```

如果定义了多重依赖关系，它们之间用“&&”间隔。依赖关系也可以应用到该菜单中所有的其他选项(同样接受 if 表达式)，下面的两段脚本是等价的：

```
bool "foo" if BAR
default y if BAR
```

和

```
depends on BAR
bool "foo"
default y
```

选择关系（也称为反向依赖关系）的格式为：

```
select <symbol> [if <expr>]
```

A 如果选择了 B，则在 A 被选中的情况下，B 自动被选中。

kbuild Makefile 中的 expr（表达式）定义为：



```
<expr> ::= <symbol>
          <symbol> '=' <symbol>
          <symbol> '!=' <symbol>
          '(' <expr> ')'
          '!' <expr>
          <expr> '&&' <expr>
          <expr> '||' <expr>
```

也就是说 `expr` 是由 `symbol`、两个 `symbol` 相等、两个 `symbol` 不等以及 `expr` 的赋值、非、与或运算构成。而 `symbol` 分为两类，一类是由菜单入口定义配置选项定义的非常数 `symbol`，另一类是作为 `expr` 组成部分的常数 `symbol`。

数据范围的格式为：

```
range <symbol> <symbol> [if <expr>]
```

为 `int` 和 `hex` 类型的选项设置可以接受输入值范围，用户只能输入大于等于第一个 `symbol`，小于等于第二个 `symbol` 的值。

帮助信息的格式为：

```
help (或---help---)
  开始
  ...
  结束
```

帮助信息完全靠文本缩进识别结束。“---help---”和“`help`”在作用上没有区别，设计“---help---”的初衷在于将文件中的配置逻辑与给开发人员的提示分开。

`menuconfig` 关键字的作用与 `config` 类似，但它在 `config` 的基础上要求所有的子选项作为独立的行显示。

(2) 菜单结构。

菜单入口在菜单树结构中的位置可由两种方法决定。第一种方式为：

```
menu "Network device support"
  depends on NET
  config NETDEVICES
  ...
endmenu
```

所有处于“`menu`”和“`endmenu`”之间的菜单入口都会成为“Network device support”的子菜单。而且，所有子菜单选项都会继承父菜单的依赖关系，比如，“Network device support”对“`NET`”的依赖会被加到了配置选项 `NETDEVICES` 的依赖列表中。

注意 `menu` 后面跟的“Network device support”项目仅仅是 1 个菜单，没有对应真实的配置选项，也不具备 3 种不同的状态。这是它和 `config` 的区别。

另一种方式是通过分析依赖关系生成菜单结构。如果菜单选项在一定程度上依赖于前面的选项，它就能成为该选项的子菜单。如果父选项为“`N`”，子选项不可见；如果父选项可见，子选项才能可见。例如：

```
config MODULES
  bool "Enable loadable module support"

config MODVERSIONS
  bool "Set version information on all module symbols"
  depends on MODULES
```



```
comment "module support disabled"
depends on !MODULES
```

MODVERSIONS 直接依赖 MODULES，只有 MODULES 不为“n”时，该选项才可见。

除此之外，Kconfig 中还可能使用“choices ... endchoice”、“comment”、“if...endif”这样的语法结构。其中“choices ... endchoice”的结构为：

```
choice
<choice options>
<choice block>
endchoice
```

它定义一个选择群，其接受的选项（choice options）可以是前面描述的任何属性，例如 LDD6410 的 VGA 输出分辨率可以是 1 024×768 或者 800×600，在 drivers/video/samsung/Kconfig 就定义了如下的 choice：

```
choice
depends on FB_S3C_VGA
prompt "Select VGA Resolution for S3C Framebuffer"
default FB_S3C_VGA_1024_768
config FB_S3C_VGA_1024_768
    bool "1 024*768@60Hz"
    ---help---
    TBA
config FB_S3C_VGA_640_480
    bool "640*480@60Hz"
    ---help---
    TBA
endchoice
```

Kconfig 配置脚本和 Makefile 脚本编写的更详细信息，可以分别参看内核文档 Documentation 目录的 kbuild 子目录下的 Kconfig-language.txt 和 Makefiles.txt 文件。

4. 应用实例：在内核中新增驱动代码目录和子目录

下面来看一个综合实例，假设我们要在内核源代码 drivers 目录下为 ARM 体系结构新增如下用于 test driver 的树型目录：

```
|--test
|  |-- cpu
|  |   |-- cpu.c
|  |-- test.c
|  |-- test_client.c
|  |-- test_ioctl.c
|  |-- test_proc.c
|  |-- test_queue.c
```

在内核中增加目录和子目录，我们需为相应的新增目录创建 Makefile 和 Kconfig 文件，而新增目录的父目录中的 Kconfig 和 Makefile 也需修改，以便新增的 Kconfig 和 Makefile 能被引用。

在新增的 test 目录下，应该包含如下 Kconfig 文件：

```
#
# TEST driver configuration
#
menu "TEST Driver "
comment " TEST Driver"

config CONFIG_TEST
    bool "TEST support "
```



```
config CONFIG_TEST_USER
    tristate "TEST user-space interface"
    depends on CONFIG_TEST

endmenu
```

由于 test driver 对于内核来说是新的功能, 所以需首先创建一个菜单 TEST Driver。然后, 显示 “TEST support”, 等待用户选择; 接下来判断用户是否选择了 TEST Driver, 如果是 (CONFIG_TEST=y), 则进一步显示子功能: 用户接口与 CPU 功能支持; 由于用户接口功能可以被编译成内核模块, 所以这里的询问语句使用了 tristate。

为了使这个 Kconfig 能起作用, 修改 arch/arm/Kconfig 文件, 增加:

```
source "drivers/test/Kconfig"
```

脚本中的 source 意味着引用新的 Kconfig 文件。

在新增的 test 目录下, 应该包含如下 Makefile 文件:

```
# drivers/test/Makefile
#
# Makefile for the TEST.
#
obj-$(CONFIG_TEST) += test.o test_queue.o test_client.o
obj-$(CONFIG_TEST_USER) += test_ioctl.o
obj-$(CONFIG_PROC_FS) += test_proc.o

obj-$(CONFIG_TEST_CPU) += cpu/
```

该脚本根据配置变量的取值, 构建 obj-*列表。由于 test 目录中包含一个子目录 cpu, 当 CONFIG_TEST_CPU=y 时, 需要将 cpu 目录加入列表。

test 目录中的 cpu 子目录也需包含如下的 Makefile:

```
# drivers/test/test/Makefile
#
# Makefile for the TEST CPU
#
obj-$(CONFIG_TEST_CPU) += cpu.o
```

为了使得整个 test 目录能够被编译命令作用到, test 目录父目录中的 Makefile 也需新增如下脚本:

```
obj-$(CONFIG_TEST) += test/
```

在 drivers/Makefile 中加入 obj-\$(CONFIG_TEST) += test/, 使得在用户在进行内核编译时能够进入 test 目录。

增加了 Kconfig 和 Makefile 之后的新的 test 树型目录为:

```
|--test
|   |-- cpu
|       |-- cpu.c
|       |-- Makefile
|   |-- test.c
|   |-- test_client.c
|   |-- test_ioctl.c
|   |-- test_proc.c
|   |-- test_queue.c
|   |-- Makefile
|   |-- Kconfig
```

3.4.3 Linux 内核的引导

引导 Linux 系统的过程包括很多阶段，这里将以引导 X86 PC 为例来进行讲解。引导 X86 PC 上的 Linux 的过程和引导嵌入式系统上的 Linux 的过程基本类似。不过在 X86 PC 上有一个从 BIOS（基本输入/输出系统）转移到 Bootloader 的过程，而嵌入式系统往往复位后就直接运行 Bootloader。

图 3.8 所示为 X86 PC 上从上电/复位到运行 Linux 用户空间初始进程的流程。在进入与 Linux 相关代码之间，会经历如下阶段。

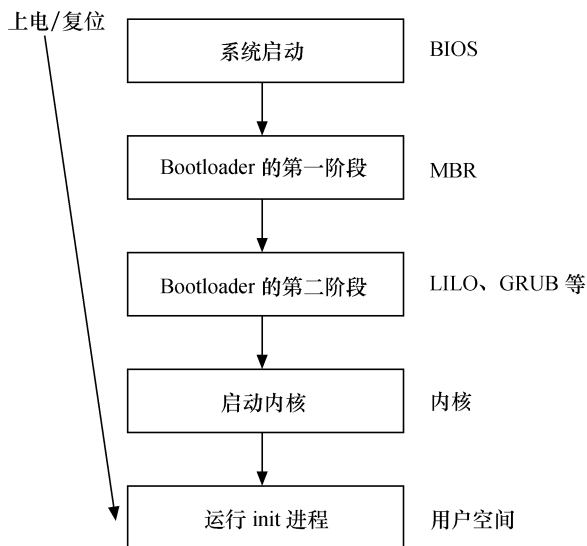


图 3.8 X86 PC 上的 Linux 引导流程

(1) 当系统上电或复位时，CPU 会将 PC 指针赋值为一个特定的地址 0xFFFF0 并执行该地址处的指令。在 PC 机中，该地址位于 BIOS 中，它保存在主板上的 ROM 或 Flash 中。

(2) BIOS 运行时按照 CMOS 的设置定义的启动设备顺序来搜索处于活动状态并且可以引导的设备。若从硬盘启动，BIOS 会将硬盘 MBR（主引导记录）中的内容加载到 RAM。MBR 是一个 512 字节大小的扇区，位于磁盘上的第一个扇区中（0 道 0 柱面 1 扇区）。当 MBR 被加载到 RAM 中之后，BIOS 就会将控制权交给 MBR。

(3) 主引导加载程序查找并加载次引导加载程序。它在分区表中查找活动分区，当找到一个活动分区时，扫描分区表中的其他分区，以确保它们都不是活动的。当这个过程验证完成之后，就将活动分区的引导记录从这个设备中读入 RAM 中并执行它。

(4) 次引导加载程序加载 Linux 内核和可选的初始 RAM 磁盘，将控制权交给 Linux 内核源代码。

(5) 运行被加载的内核，并启动用户空间应用程序。

嵌入式系统中 Linux 的引导过程与之类似，但一般更加简洁。不论具体以怎样的方式实现，只要具备如下特征就可以称其为 Bootloader。

- 可以在系统上电或复位的时候以某种方式执行，这些方式包括被 BIOS 引导执行、直接在 NOR Flash 中执行、NAND Flash 中的代码被 MCU 自动拷入内部或外部 RAM 执行等。
- 能将 U 盘、磁盘、光盘、NOR/NAND Flash、ROM、SD 卡等存储介质，甚或网口、串



口中的操作系统加载到 RAM 并把控制权交给操作系统源代码执行。

完成上述功能的 Bootloader 的实现方式非常多样化, 甚至本身也可以是一个简化版的操作系统。著名的 Linux Bootloader 包括应用于 PC 的 LILO 和 GRUB, 应用于嵌入式系统的 U-Boot、RedBoot 等。

相比较于 LILO, GRUB 本身能理解 EXT2、EXT3 文件系统, 因此可在文件系统中加载 Linux, 而 LILO 只能识别“裸扇区”。

U-Boot 的定位为“Universal Bootloader”, 其功能比较强大, 涵盖了包括 PowerPC、ARM、MIPS 和 X86 在内的绝大部分处理器构架, 提供网卡、串口、Flash 等外设驱动, 提供必要的网络协议 (BOOTP、DHCP、TFTP), 能识别多种文件系统 (cramfs、fat、jffs2 和 registerfs 等), 并附带了调试、脚本、引导等工具, 应用十分广泛。

Redboot 是 Redhat 公司随 eCos 发布的 Bootloader 开源项目, 除了包含 U-Boot 类似的强大功能外, 它还包含 GDB stub (插桩), 因此能通过串口或网口与 GDB 进行通信, 调试 GCC 产生的任何程序 (包括内核)。

我们有必要对上述流程的第 5 个阶段进行更详细的分析, 它完成启动内核并运行用户空间的 init 进程。

当内核映像被加载到 RAM 之后, Bootloader 的控制权被释放, 内核阶段就开始了。内核映像并不是完全可直接执行的目标代码, 而是一个压缩过的 zImage (小内核) 或 bzImage (大内核, bzImage 中的 b 是“big”的意思)。

但是, 并非 zImage 和 bzImage 映像中的一切都被压缩了, 否则 Bootloader 把控制权交给这个内核映像它就“傻”了。实际上, 映像中包含未被压缩的部分, 这部分中包含解压缩程序, 解压缩程序会解压映像中被压缩的部分。zImage 和 bzImage 都是用 gzip 压缩的, 它们不仅是一个压缩文件, 而且在这两个文件的开头部分内嵌有 gzip 解压缩代码。

如图 3.9 所示, 当 bzImage (用于 i386 映像) 被调用时, 它从/arch/i386/boot/head.S 的 start 汇编例程开始执行。这个程序执行一些基本的硬件设置, 并调用/arch/i386/boot/compressed/head.S 中的 startup_32 例程。startup_32 程序设置一些基本的运行环境 (如堆栈) 后, 清除 BSS 段, 调用/arch/i386/boot/compressed/misc.c 中的 decompress_kernel() C 函数解压内核。内核被解压到内存中之后, 会再调用/arch/i386/kernel/head.S 文件中的 startup_32 例程, 这个新的 startup_32 例程 (称为清除

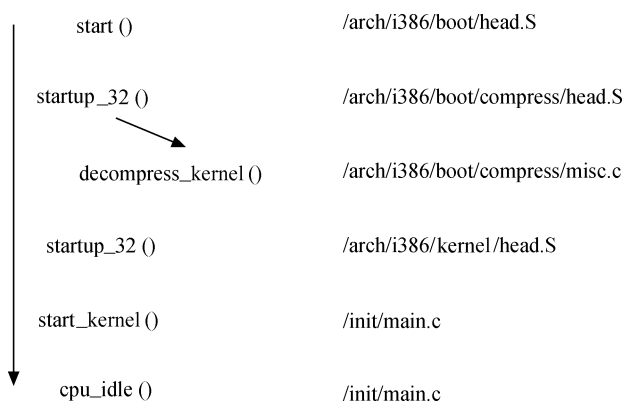


图 3.9 X86 PC 上的 Linux 内核初始化

程序或进程 0) 会初始化页表, 并启用内存分页机制, 接着为任何可选的浮点单元 (FPU) 检测 CPU 的类型, 并将其存储起来供以后使用。这些都做完之后, /init/main.c 中的 start_kernel() 函数被调用, 进入与体系结构无关的 Linux 内核部分。

start_kernel() 会调用一系列初始化函数来设置中断, 执行进一步的内存配置。之后, /arch/i386/kernel/process.c 中 kernel_thread() 被调用以启动第一个核心线程, 该线程执行 init() 函数, 而原执行序列会调用 cpu_idle() 等待调度。

作为核心线程的 init() 函数完成外设及其驱动程序的加载和初始化, 挂接根文件系统。init() 打开 /dev/console 设备, 重定向 stdin、stdout 和 stderr 到控制台。之后, 它搜索文件系统中的 init 程序 (也可以由 “init=” 命令行参数指定 init 程序), 并使用 execve() 系统调用执行 init 程序。搜索 init 程序的顺序为: /sbin/init、/etc/init、/bin/init 和 /bin/sh。在嵌入式系统中, 多数情况下, 可以给内核传入一个简单的 shell 脚本来启动必需的嵌入式应用程序。

至此, 漫长的 Linux 内核引导和启动过程就此结束, 而 init() 对应的这个由 start_kernel() 创建的第一个线程也进入用户模式。

3.5 Linux 下的 C 编程特点

3.5.1 Linux 编码风格

Linux 程序的命名习惯和 Windows 程序的命名习惯及著名的匈牙利命名法有很大的不同。

在 Windows 程序中, 习惯以如下方式命名宏、变量和函数:

```
#define PI 3.141 592 6 /*用大写字母代表宏*/
int min_value, max_value; /*变量: 第一个单词全写, 其后的单词第一个字母小写*/
void SendData(void); /*函数: 所有单词第一个字母都大写定义*/
```

这种命名方式在程序员中非常盛行, 意思表达清晰且避免了匈牙利法的臃肿, 单词之间通过首字母大写来区分。通过第 1 个单词的首字母是否大写可以区分名称属于变量还是属于函数, 而看到整串的大写字母可以断定为宏。实际上, Windows 的命名习惯并非仅限于 Windows 编程, 大多数领域的程序开发都遵照此习惯。

但是 Linux 不以这种习惯命名, 对应于上面的一段程序, 在 Linux 中会被命名为:

```
#define PI 3.141 592 6
int min_value, max_value;
void send_data(void);
```

上述命名方式中, 下划线大行其道, 不依照 Windows 所采用的首字母大写以区分单词的方式。Linux 的命名习惯与 Windows 命名习惯各有千秋, 但是既然本书和本书的读者立足于编写 Linux 程序, 代码风格理应保持与 Linux 开发社区的一致性。

Linux 的代码缩进使用 “TAB” (8 个字符)。

Linux 的代码括号 “{” 和 “}” 的使用原则如下。

(1) 对于结构体、if/for/while/switch 语句, “{” 不另起一行, 例如:

```
struct var_data {
    int len;
    char data[0];
```



```
};

if (a == b) {
    a = c;
    d = a;
}

for (i = 0; i < 10; i++) {
    a = c;
    d = a;
}
```

(2) 如果 if、for 循环后只有 1 行，不要加 “{” 和 “}”，例如：

```
for (i = 0; i < 10; i++) {
    a = c;
}
```

应该改为：

```
for (i = 0; i < 10; i++)
    a = c;
```

(3) if 和 else 混用的情况下，else 语句不另起一行，例如：

```
if (x == y) {
    ...
} else if (x > y) {
    ...
} else {
    ...
}
```

(4) 对于函数，“{” 另起一行，譬如：

```
int add(int a, int b)
{
    return a + b;
}
```

在 switch/case 语句方面，Linux 建议 switch 和 case 对齐，例如：

```
switch (suffix) {
case 'G':
case 'g':
    mem <= 30;
    break;
case 'M':
case 'm':
    mem <= 20;
    break;
case 'K':
case 'k':
    mem <= 10;
    /* fall through */
default:
    break;
}
```

内核下的 Documentation/CodingStyle 描述了 Linux 内核对编码风格的要求，内核下的 scripts/checkpatch.pl 提供了 1 个检查代码风格的脚本。如果我们使用 scripts/checkpatch.pl 检查包含如下代码块的源程序：

```
for (i = 0; i < 10; i++) {
    a = c;
}
```

就会产生“WARNING: braces {} are not necessary for single statement blocks”的警告。

另外，请注意代码中空格的运用，譬如“for(i=0;i<10;i++){”语句中“ ”都是空格。

3.5.2 GNU C 与 ANSI C

Linux 上可用的 C 编译器是 GNU C 编译器，它建立在自由软件基金会的编程许可证的基础上，因此可以自由发布。GNU C 对标准 C 进行一系列扩展，以增强标准 C 的功能。

1. 零长度和变量长度数组

GNU C 允许使用零长度数组，在定义变长对象的头结构时，这个特性非常有用。例如：

```
struct var_data {
    int len;
    char data[0];
};
```

char data[0]仅仅意味着程序中通过 var_data 结构体实例的 data[index]成员可以访问 len 之后的第 index 个地址，它并没有为 data[]数组分配内存，因此 sizeof(struct var_data)=sizeof(int)。

假设 struct var_data 的数据域就保存在 struct var_data 紧接着的内存区域，则通过如下代码可以遍历这些数据：

```
struct var_data s;
...
for (i = 0; i < s.len; i++)
    printf("%02x", s.data[i]);
```

GNU C 中也可以使用 1 个变量定义数组，例如如下代码中定义的“double x[n]”：

```
int main (int argc, char *argv[])
{
    int i, n = argc;
    double x[n];

    for (i = 0; i < n; i++)
        x[i] = i;

    return 0;
}
```

2. case 范围

GNU C 支持 case x...y 这样的语法，区间[x,y]的数都会满足这个 case 的条件，请看下面的代码：

```
switch (ch) {
case '0'... '9': c -= '0';
    break;
case 'a'... 'f': c -= 'a' - 10;
    break;
case 'A'... 'F': c -= 'A' - 10;
    break;
}
```

代码中的 case '0'... '9'等价于标准 C 中的：

```
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
```



3. 语句表达式

GNU C 把包含在括号中的复合语句看做是一个表达式,称为语句表达式,它可以出现在任何允许表达式的地方。我们可以在语句表达式中使用原本只能在复合语句中使用的循环、局部变量等,例如:

```
#define min_t(type,x,y) \
({ type __x = (x); type __y = (y); __x < __y ? __x: __y; })

int ia, ib, mini;
float fa, fb, minf;

mini = min_t(int, ia, ib);
minf = min_t(float, fa, fb);
```

因为重新定义了__xx和__y这两个局部变量,所以以上述方式定义的宏将不会有副作用。在标准 C 中,对应的如下宏则会产生副作用:

```
#define min(x,y) ((x) < (y) ? (x) : (y))
```

代码 min(++ia,++ib)会被展开为((++ia) < (++ib) ? (++ia): (++ib)),传入宏的“参数”被增加 2 次。

4. typeof 关键字

typeof(x)语句可以获得 x 的类型,因此,我们可以借助 typeof 重新定义 min 这个宏:

```
#define min(x,y) ({ \
    const typeof(x) _x = (x);          \
    const typeof(y) _y = (y);          \
    (void) (&_x == &_y);               \
    _x < _y ? _x : _y; })
```

我们不需要像 min_t(type,x,y)这个宏那样把 type 传入,因为通过 typeof(x)、typeof(y)可以获得 type。代码行(void) (&_x == &_y)的作用是检查_x和_y的类型是否一致。

5. 可变参数宏

标准 C 就支持可变参数函数,意味着函数的参数是不固定的,例如 printf()函数的原型为:

```
int printf( const char *format [, argument]... );
```

而在 GNU C 中,宏也可以接受可变数目的参数,例如:

```
#define pr_debug(fmt,arg...) \
    printk(fmt,##arg)
```

这里 arg 表示其余的参数,可以是零个或多个,这些参数以及参数之间的逗号构成 arg 的值,在宏扩展时替换 arg,例如下列代码:

```
pr_debug("%s:%d",filename,line)
```

会被扩展为:

```
printk("%s:%d", filename, line)
```

使用“##”的原因是处理 arg 不代表任何参数的情况,这时候,前面的逗号就变得多余了。

使用“##”之后,GNU C 预处理器会丢弃前面的逗号,这样,代码:

```
pr_debug("success!\n")
```

会被正确地扩展为:

```
printk("success!\n")
```

而不是:

```
printk("success!\n",)
```

这正是我们希望看到的。

6. 标号元素

标准 C 要求数组或结构体的初始化值必须以固定的顺序出现，在 GNU C 中，通过指定索引或结构体成员名，允许初始化值以任意顺序出现。

指定数组索引的方法是在初始化值前添加 “[INDEX]=”，当然也可以用 “[FIRST ... LAST]=” 的形式指定一个范围。例如，下面的代码定义一个数组，并把其中的所有元素赋值为 0：

```
unsigned char data[MAX] = { [0 ... MAX-1] = 0 };
```

下面的代码借助结构体成员名初始化结构体：

```
struct file_operations ext2_file_operations = {
    llseek: generic_file_llseek,
    read: generic_file_read,
    write: generic_file_write,
    ioctl: ext2_ioctl,
    mmap: generic_file_mmap,
    open: generic_file_open,
    release: ext2_release_file,
    fsync: ext2_sync_file,
};
```

但是，Linux 2.6 推荐类似的代码应该尽量采用标准 C 的方式：

```
struct file_operations ext2_file_operations = {
    .llseek      = generic_file_llseek,
    .read        = generic_file_read,
    .write       = generic_file_write,
    .aio_read    = generic_file_aio_read,
    .aio_write   = generic_file_aio_write,
    .ioctl       = ext2_ioctl,
    .mmap        = generic_file_mmap,
    .open        = generic_file_open,
    .release     = ext2_release_file,
    .fsync       = ext2_sync_file,
    .readv       = generic_file_readv,
    .writev      = generic_file_writev,
    .sendfile    = generic_file_sendfile,
};
```

7. 当前函数名

GNU C 预定义了两个标志符保存当前函数的名字，`__FUNCTION__` 保存函数在源码中的名字，`__PRETTY_FUNCTION__` 保存带语言特色的名字。在 C 函数中，这两个名字是相同的。

```
void example()
{
    printf("This is function:%s", __FUNCTION__);
}
```

代码中的 `__FUNCTION__` 意味着字符串 “example”。C99 已经支持 `__func__` 宏，因此建议在 Linux 编程中不再使用 `__FUNCTION__`，而转而使用 `__func__`：

```
void example()
{
    printf("This is function:%s", __func__);
}
```

8. 特殊属性声明

GNU C 允许声明函数、变量和类型的特殊属性，以便进行手工的代码优化和定制代码检查的方法。要指定一个声明的属性，只需要在声明后添加 `__attribute__((ATTRIBUTE))`。其中



ATTRIBUTE 为属性说明, 如果存在多个属性, 则以逗号分隔。GNU C 支持 `noreturn`、`format`、`section`、`aligned`、`packed` 等十多个属性。

`noreturn` 属性作用于函数, 表示该函数从不返回。这会让编译器优化代码, 并消除不必要的警告信息。例如:

```
# define ATTRIB_NORET __attribute__((noreturn)) ....
asmlinkage NORET_TYPE void do_exit(long error_code) ATTRIB_NORET;
```

`format` 属性也用于函数, 表示该函数使用 `printf`、`scanf` 或 `strftime` 风格的参数, 指定 `format` 属性可以让编译器根据格式串检查参数类型。例如:

```
asmlinkage int printk(const char * fmt, ...) __attribute__((format (printf, 1, 2)));
```

上述代码中的第 1 个参数是格式串, 从第 2 个参数开始都会根据 `printf()` 函数的格式串规则检查参数。

`unused` 属性作用于函数和变量, 表示该函数或变量可能不会被用到, 这个属性可以避免编译器产生警告信息。

`aligned` 属性用于变量、结构体或联合体, 指定变量、结构体或联合体的对界方式, 以字节为单位, 例如:

```
struct example_struct {
    char a;
    int b;
    long c;
} __attribute__((aligned(4)));
```

表示该结构类型的变量以 4 字节对界。

`packed` 属性作用于变量和类型, 用于变量或结构体成员时表示使用最小可能的对界, 用于枚举、结构体或联合体类型时表示该类型使用最小的内存。例如:

```
struct example_struct {
    char a;
    int b;
    long c __attribute__((packed));
};
```



编译器对结构体成员及变量对界的目的是为了更快地访问结构体成员及变量占据的内存。例如, 对于一个 32 位的整型变量, 若以 4 字节方式存放 (即低两位地址为 00), 则 CPU 在一个总线周期内就可以读取 32 位; 若不然, CPU 需要两次总线周期才能组合为一个 32 位整型。

9. 内建函数

GNU C 提供了大量的内建函数, 其中大部分是标准 C 库函数的 GNU C 编译器内建版本, 例如 `memcpy()` 等, 它们与对应的标准 C 库函数功能相同。

不属于库函数的其他内建函数的命名通常以 `__builtin` 开始, 如下所示。

- 内建函数 `__builtin_return_address (LEVEL)` 返回当前函数或其调用者的返回地址, 参数 `LEVEL` 指定调用栈的级数, 如 0 表示当前函数的返回地址, 1 表示当前函数的调用者的返回地址。
- 内建函数 `__builtin_constant_p(EXP)` 用于判断一个值是否为编译时常数, 如果参数 `EXP` 的值是常数, 函数返回 1, 否则返回 0。
- 内建函数 `__builtin_expect(EXP, C)` 用于为编译器提供分支预测信息, 其返回值是整数表达

式 EXP 的值，C 的值必须是编译时常数。

例如，下面的代码检测第 1 个参数是否为编译时常数以确定采用参数版本还是非参数版本的代码：

```
#define test_bit(nr,addr) \
    (_builtin_constant_p(nr) ? \
    constant_test_bit((nr),(addr)) : \
    variable_test_bit((nr),(addr)))
```

在使用 gcc 编译 C 程序的时候，如果使用“-ansi -pedantic”编译选项，则会告诉编译器不使用 GNU 扩展语法。例如对于如下 C 程序 test.c：

```
struct var_data {
    int len;
    char data[0];
};

struct var_data a;
```

直接编译可以通过：

```
gcc -c test.c
```

如果使用“-ansi -pedantic”编译选项，编译会报警：

```
gcc -ansi -pedantic -c test.c
test.c:3: warning: ISO C forbids zero-size array 'data'
```

3.5.3 do {} while(0)

在 Linux 内核中，经常会看到 do {} while(0) 这样的语句，许多人开始都会疑惑，认为 do {} while(0) 毫无意义，因为它只会执行一次，加不加 do {} while(0) 效果是完全一样的，其实 do {} while(0) 的用法主要用于宏定义中。

这里用一个简单点的宏来演示：

```
#define SAFE_FREE(p) do{ free(p); p = NULL;} while(0)
```

假设这里去掉 do...while(0)，即定义 SAFE_DELETE 为：

```
#define SAFE_FREE(p) free(p); p = NULL;
```

那么以下代码

```
if(NULL != p)
    SAFE_DELETE(p)
else
    .../* do something */
```

会被展开为：

```
if(NULL != p)
    free(p); p = NULL;
else
    .../* do something */
```

展开的代码中存在两个问题。

- (1) 因为 if 分支后有两个语句，导致 else 分支没有对应的 if，编译失败。
- (2) 假设没有 else 分支，则 SAFE_FREE 中的第二个语句无论 if 测试是否通过，都会执行。的确，将 SAFE_FREE 的定义加上 {} 就可以解决上述问题了，即：

```
#define SAFE_FREE(p) { free(p); p = NULL;}
```

这样，代码：



```
if(NULL != p)
    SAFE_DELETE(p)
else
    ... /* do something */
```

会被展开为:

```
if(NULL != p)
    { free(p); p = NULL; }
else
    ... /* do something */
```

但是, 在 C 程序中, 每个语句后面加分号是一种约定俗成的习惯, 那么, 如下代码:

```
if(NULL != p)
    SAFE_DELETE(p);
else
    ... /* do something */
```

将被扩展为:

```
if(NULL != p)
    { free(p); p = NULL; };
else
    ... /* do something */
```

这样, `else` 分支就又没有对应的 `if` 了, 编译将无法通过。假设用了 `do {} while(0)`, 情况就不一样了, 同样的代码会被展开为:

```
if(NULL != p)
    do{ free(p); p = NULL; } while(0);
else
    ... /* do something */
```

不会再出现编译问题。`do while(0)` 的使用完全是为了保证宏定义的使用者能无编译错误地使用宏, 它不对其使用者做任何假设。

3.5.4 goto

用不用 `goto` 一直是一个著名的争议话题, Linux 内核源代码中对 `goto` 的应用非常广泛, 但是一般只限于错误处理中, 其结构如:

```
if(register_a() != 0)
    goto err;
if(register_b() != 0)
    goto err1;
if(register_c() != 0)
    goto err2;
if(register_d() != 0)
    goto err3;

...

err3:
    unregister_c();
err2:
    unregister_b();
err1:
    unregister_a();
err:
    return ret;
```

这种 goto 用于错误处理的用法实在是简单而高效，只需保证在错误处理时注销、资源释放等与正常的注册、资源申请顺序相反。

3.6 总结

本章主要讲解了 Linux 内核和 Linux 内核编程的基础知识，为进行 Linux 驱动开发打下软件基础。

在 Linux 内核方面，主要介绍了 Linux 内核的发展史、组成、特点、源代码结构、内核编译方法及内核引导过程。

由于 Linux 驱动编程本质属于内核编程，因此掌握内核编程的基础知识显得尤为重要。本章在这方面主要讲解了在内核中新增程序及目录和编写 Kconfig 和 Makefile 的方法，并分析了 Linux 下 C 编程习惯以及 Linux 所使用的 GNU C 针对标准 C 的扩展语法。

LINUX

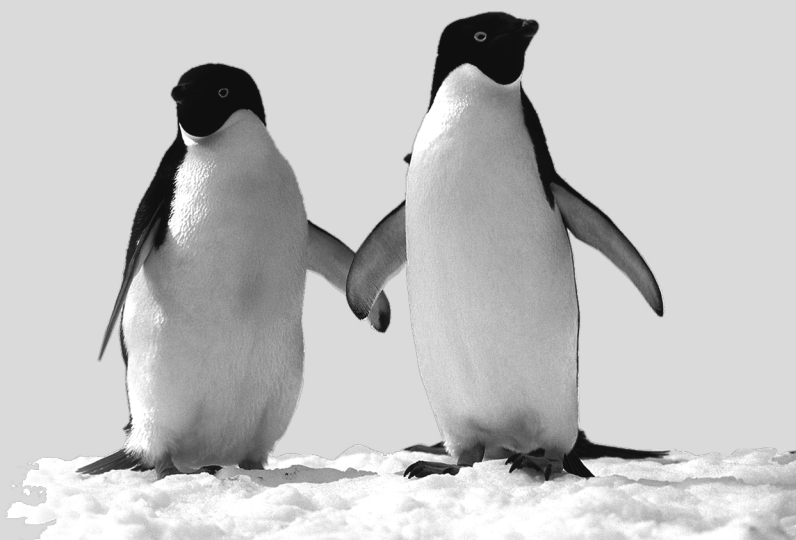
第4章 Linux 内核模块

本章导读

Linux 设备驱动会以内核模块的形式出现，因此，学会编写 Linux 内核模块编程是学习 Linux 设备驱动的先决条件。

4.1~4.2 节讲解了 Linux 内核模块的概念和结构，4.3~4.8 节对 Linux 内核模块的各个组成部分进行了展现，4.1~4.2 节与 4.3~4.8 节是整体与部分的关系。

4.9 节说明了独立存在的 Linux 内核模块的 Makefile 文件编写方法和模块的编译方法。



4.1 Linux 内核模块简介

Linux 内核的整体结构已经非常庞大，而其包含的组件也非常多。我们怎样把需要的部分都包含在内核中呢？

一种方法是把所有需要的功能都编译到 Linux 内核。这会导致两个问题，一是生成的内核会很大，二是如果我们要在现有的内核中新增或删除功能，将不得不重新编译内核。

有没有一种机制使得编译出的内核本身并不需要包含所有功能，而在这些功能需要被使用的时候，其对应的代码被动态地加载到内核中呢？

Linux 提供了这样的一种机制，这种机制被称为模块（Module）。模块具有这样的特点。

- 模块本身不被编译入内核映像，从而控制了内核的大小。
- 模块一旦被加载，它就和内核中的其他部分完全一样。

为了使读者建立对模块的初步感性认识，我们先来看一个最简单的内核模块“Hello World”，如代码清单 4.1 所示。

代码清单 4.1 一个最简单的 Linux 内核模块

```
1 #include <linux/init.h>
2 #include <linux/module.h>
3
4 static int hello_init(void)
5 {
6     printk(KERN_INFO " Hello World enter\n");
7     return 0;
8 }
9
10 static void hello_exit(void)
11 {
12     printk(KERN_INFO " Hello World exit\n ");
13 }
14
15 module_init(hello_init);
16 module_exit(hello_exit);
17
18 MODULE_AUTHOR("Barry Song <21cnbao@gmail.com>");
19 MODULE_LICENSE("Dual BSD/GPL");
20 MODULE_DESCRIPTION("A simple Hello World Module");
21 MODULE_ALIAS("a simplest module");
```

这个最简单的内核模块只包含内核模块加载函数、卸载函数和对 Dual BSD/GPL 许可权限的声明以及一些描述信息，位于本书配套光盘 VirtualBox 虚拟机映像的/home/lihacker/develop/svn/1dd6410-read-only/training/kernel/drivers/hello 目录。编译它会产生 hello.ko 目标文件，通过“insmod ./hello.ko”命令可以加载它，通过“rmmod hello”命令可以卸载它，加载时输出“Hello World enter”，卸载时输出“Hello World exit”。

内核模块中用于输出的函数是内核空间的 printk()而非用户空间的 printf()，printk()的用法和 printf()基本相似，但前者可定义输出级别。printk()可作为一种最基本的内核调试手段，在 Linux



驱动的调试章节中将详细讲解这个函数。

在 Linux 中, 使用 `lsmod` 命令可以获得系统中加载了的所有模块以及模块间的依赖关系, 例如:

```
Module              Size Used by
hello                9 472  0
nls_iso8859_1       12 032  1
nls_cp437           13 696  1
vfat                18 816  1
fat                 57 376  1 vfat
...
```

`lsmod` 命令实际上读取并分析 “`/proc/modules`” 文件, 与上述 `lsmod` 命令结果对应的 “`/proc/modules`” 文件如下:

```
lihacker@lihacker-laptop:~/ $ cat /proc/modules
hello 9472 0 - Live 0xf953b000
nls_iso8859_1 12032 1 - Live 0xf950c000
nls_cp437 13696 1 - Live 0xf9561000
vfat 18816 1 - Live 0xf94f3000
...
```

内核中已加载模块的信息也存在于 `/sys/module` 目录下, 加载 `hello.ko` 后, 内核中将包含 `/sys/module/hello` 目录, 该目录下又包含一个 `refcnt` 文件和一个 `sections` 目录, 在 `/sys/module/hello` 目录下运行 “`tree -a`” 得到如下目录树:

```
lihacker@lihacker-laptop:/sys/module/hello$ tree -a
.
|-- holders
|-- initstate
|-- notes
|   '-- .note.gnu.build-id
|-- refcnt
|-- sections
|   |-- .bss
|   |-- .data
|   |-- .gnu.linkonce.this_module
|   |-- .note.gnu.build-id
|   |-- .rodata.str1.1
|   |-- .strtab
|   |-- .symtab
|   '-- .text
'-- srcversion

3 directories, 12 files
```

`modprobe` 命令比 `insmod` 命令要强大, 它在加载某模块时, 会同时加载该模块所依赖的其他模块。使用 `modprobe` 命令加载的模块若以 “`modprobe -r filename`” 的方式卸载将同时卸载其依赖的模块。

使用 `modinfo <模块名>` 命令可以获得模块的信息, 包括模块作者、模块的说明、模块所支持的参数以及 `vermagic`:

```
lihacker@lihacker-laptop: ~ /develop/svn/ldd6410-read-only/training/kernel/drivers/
hello$ modinfo hello.ko
filename:          hello.ko
alias:             a simplest module
description:       A simple Hello World Module
license:           Dual BSD/GPL
```



```
author:      Barry Song <21cnbao@gmail.com>
srcversion:  3FE9B0FBAFDD565399B9C05
depends:
vermagic:    2.6.28-11-generic SMP mod_unload modversions 586
```

4.2 Linux 内核模块程序结构

一个 Linux 内核模块主要由如下几个部分组成。

(1) 模块加载函数（一般需要）。

当通过 `insmod` 或 `modprobe` 命令加载内核模块时，模块的加载函数会自动被内核执行，完成本模块的相关初始化工作。

(2) 模块卸载函数（一般需要）。

当通过 `rmmod` 命令卸载某模块时，模块的卸载函数会自动被内核执行，完成与模块卸载函数相反的功能。

(3) 模块许可证声明（必须）。

许可证（`LICENSE`）声明描述内核模块的许可权限，如果不声明 `LICENSE`，模块被加载时，将收到内核被污染（`kernel tainted`）的警告。

在 Linux 2.6 内核中，可接受的 `LICENSE` 包括“`GPL`”、“`GPL v2`”、“`GPL and additional rights`”、“`Dual BSD/GPL`”、“`Dual MPL/GPL`”和“`Proprietary`”。

大多数情况下，内核模块应遵循 `GPL` 兼容许可权。Linux 2.6 内核模块最常见的是以 `MODULE_LICENSE("Dual BSD/GPL")` 语句声明模块采用 `BSD/GPL` 双 `LICENSE`。

(4) 模块参数（可选）。

模块参数是模块被加载的时候可以被传递给它的值，它本身对应模块内部的全局变量。

(5) 模块导出符号（可选）。

内核模块可以导出符号（`symbol`，对应于函数或变量），这样其他模块可以使用本模块中的变量或函数。

(6) 模块作者等信息声明（可选）。

4.3 模块加载函数

Linux 内核模块加载函数一般以 `_init` 标识声明，典型的模块加载函数的形式如代码清单 4.2 所示。

代码清单 4.2 内核模块加载函数

```
1 static int __init initialization_function(void)
2 {
3     /* 初始化代码 */
4 }
5 module_init(initialization_function);
```



模块加载函数必须以“`module_init(函数名)`”的形式被指定。它返回整型值，若初始化成功，应返回 0。而在初始化失败时，应该返回错误编码。在 Linux 内核里，错误编码是一个负值，在 `<linux/errno.h>` 中定义，包含 `-ENODEV`、`-ENOMEM` 之类的符号值。总是返回相应的错误编码是种非常好的习惯，因为只有这样，用户程序才可以利用 `perror` 等方法把它们转换成有意义的错误信息字符串。

在 Linux 2.6 内核中，可以使用 `request_module(const char *fmt, ...)` 函数加载内核模块，驱动开发人员可以通过调用

```
request_module(module_name);
```

或

```
request_module("char-major-%d-%d", MAJOR(dev), MINOR(dev));
```

这种灵活的方式加载其他内核模块。

在 Linux 中，所有标识为 `__init` 的函数在连接的时候都放在 `.init.text` 这个区段内，此外，所有的 `__init` 函数在区段 `.initcall.init` 中还保存了一份函数指针，在初始化时内核会通过这些函数指针调用这些 `__init` 函数，并在初始化完成后，释放 `init` 区段（包括 `.init.text`、`.initcall.init` 等）。

4.4 模块卸载函数

Linux 内核模块加载函数一般以 `__exit` 标识声明，典型的模块卸载函数的形式如代码清单 4.3 所示。

代码清单 4.3 内核模块卸载函数

```
1 static void __exit cleanup_function(void)
2 {
3     /* 释放代码 */
4 }
5 module_exit(cleanup_function);
```

模块卸载函数在模块卸载的时候执行，不返回任何值，必须以“`module_exit(函数名)`”的形式来指定。

通常来说，模块卸载函数要完成与模块加载函数相反的功能，如下所示。

- 若模块加载函数注册了 XXX，则模块卸载函数应该注销 XXX。
- 若模块加载函数动态申请了内存，则模块卸载函数应释放该内存。
- 若模块加载函数申请了硬件资源（中断、DMA 通道、I/O 端口和 I/O 内存等）的占用，则模块卸载函数应释放这些硬件资源。
- 若模块加载函数开启了硬件，则卸载函数中一般要关闭之。

和 `__init` 一样，`__exit` 也可以使对应函数在运行完成后自动回收内存。实际上，`__init` 和 `__exit` 都是宏，其定义分别为：

```
#define __init __attribute__((__section__(".init.text")))
```

和

```
#ifndef MODULE
#define __exit __attribute__((__section__(".exit.text")))
#else
```

```
#define __exit      __attribute_used__ __attribute__((__section__(".exit.text")))
#endif
```

数据也可以被定义为 `__initdata` 和 `__exitdata`，这两个宏分别为：

```
#define __initdata  __attribute__((__section__(".init.data")))
```

和

```
#define __exitdata  __attribute__((__section__(".exit.data")))
```

4.5 模块参数

我们可以用 “`module_param(参数名,参数类型,参数读/写权限)`” 为模块定义一个参数，例如下列代码定义了 1 个整型参数和 1 个字符指针参数：

```
static char *book_name = "dissecting Linux Device Driver ";
static int num = 4 000;
module_param(num, int, S_IRUGO);
module_param(book_name, charp, S_IRUGO);
```

在装载内核模块时，用户可以向模块传递参数，形式为 “`insmode（或 modprobe）模块名 参数名=参数值`”，如果不传递，参数将使用模块内定义的缺省值。

参数类型可以是 `byte`、`short`、`ushort`、`int`、`uint`、`long`、`ulong`、`charp`（字符指针）、`bool` 或 `invbool`（布尔的反），在模块被编译时会将 `module_param` 中声明的类型与变量定义的类型进行比较，判断是否一致。

模块被加载后，在 `/sys/module/` 目录下将出现以此模块名命名的目录。当 “参数读/写权限” 为 0 时，表示此参数不存在 `sysfs` 文件系统下对应的文件节点，如果此模块存在 “参数读/写权限” 不为 0 的命令行参数，在此模块的目录下还将出现 `parameters` 目录，包含一系列以参数名命名的文件节点，这些文件的权限值就是传入 `module_param()` 的 “参数读/写权限”，而文件的内容为参数的值。

除此之外，模块也可以拥有参数数组，形式为 “`module_param_array(数组名,数组类型,数组长,参数读/写权限)`”。从 2.6.0~2.6.10 版本，需将数组长变量名赋给 “数组长”，从 2.6.10 版本开始，需将数组长变量的指针赋给 “数组长”，当不需要保存实际输入的数组元素个数时，可以设置 “数组长” 为 `NULL`。

运行 `insmod` 或 `modprobe` 命令时，应使用逗号分隔输入的数组元素。

现在我们定义一个包含两个参数的模块（如代码清单 4.4，位于虚拟机 `/home/lihacker/develop/svn/ldd6410-read-only/training/kernel/drivers/param` 目录），并观察模块加载时被传递参数和不传递参数时的输出。

代码清单 4.4 带参数的内核模块

```
1 #include <linux/init.h>
2 #include <linux/module.h>
3 MODULE_LICENSE("Dual BSD/GPL");
4
5 static char *book_name = "dissecting Linux Device Driver";
6 static int num = 4 000;
7
8 static int book_init(void)
9 {
10     printk(KERN_INFO " book name:%s\n",book_name);
```



```
11     printk(KERN_INFO " book num:%d\n",num);
12     return 0;
13 }
14 static void book_exit(void)
15 {
16     printk(KERN_INFO " Book module exit\n ");
17 }
18 module_init(book_init);
19 module_exit(book_exit);
20 module_param(num, int, S_IRUGO);
21 module_param(book_name, charp, S_IRUGO);
22
23 MODULE_AUTHOR("Barry Song <21cnbao@gmail.com>");
24 MODULE_DESCRIPTION("A simple Module for testing module params");
25 MODULE_VERSION("V1.0");
```

对上述模块运行“`insmod book.ko`”命令加载，相应输出都为模块内的默认值，通过查看“`/var/log/messages`”日志文件可以看到内核的输出：

```
[root@localhost driver_study]# tail -n 2 /var/log/messages
Jul  2 01:03:10 localhost kernel: <6> book name:dissecting Linux Device Driver
Jul  2 01:03:10 localhost kernel:  book num:4 000
```

当用户运行“`insmod book.ko book_name='GoodBook' num=5 000`”命令时，输出的是用户传递的参数：

```
[root@localhost driver_study]# tail -n 2 /var/log/messages
Jul  2 01:06:21 localhost kernel: <6> book name:GoodBook
Jul  2 01:06:21 localhost kernel:  book num:5 000
```

4.6 导出符号

Linux 2.6 的“`/proc/kallsyms`”文件对应着内核符号表，它记录了符号以及符号所在的内存地址。

模块可以使用如下宏导出符号到内核符号表：

```
EXPORT_SYMBOL(符号名);
EXPORT_SYMBOL_GPL(符号名);
```

导出的符号将可以被其他模块使用，使用前声明一下即可。`EXPORT_SYMBOL_GPL()`只适用于包含 GPL 许可权的模块。代码清单 4.5 给出了一个导出整数加、减运算函数符号的内核模块的例子（这些导出符号毫无实际意义，仅仅是为了演示）。

代码清单 4.5 内核模块中的符号导出

```
1 #include <linux/init.h>
2 #include <linux/module.h>
3 MODULE_LICENSE("Dual BSD/GPL");
4
5 int add_integar(int a,int b)
6 {
7     return a+b;
8 }
9
10 int sub_integar(int a,int b)
11 {
```

```

12 return a-b;
13 }
14
15 EXPORT_SYMBOL(add_integar);
16 EXPORT_SYMBOL(sub_integar);

```

从“/proc/kallsyms”文件中找出 add_integar、sub_integar 相关信息:

```

[root@localhost driver_study]# cat /proc/kallsyms | grep integar
c886f050 r __kcrctab_add_integar      [export]
c886f058 r __kstrtab_add_integar      [export]
c886f070 r __ksymtab_add_integar      [export]
c886f054 r __kcrctab_sub_integar      [export]
c886f064 r __kstrtab_sub_integar      [export]
c886f078 r __ksymtab_sub_integar      [export]
c886f000 T add_integar [export]
c886f00b T sub_integar [export]
13db98c9 a __crc_sub_integar [export]
e1626dee a __crc_add_integar [export]

```

4.7 模块声明与描述

在 Linux 内核模块中，我们可以用 MODULE_AUTHOR、MODULE_DESCRIPTION、MODULE_VERSION、MODULE_DEVICE_TABLE、MODULE_ALIAS 分别声明模块的作者、描述、版本、设备表和别名，例如：

```

MODULE_AUTHOR(author);
MODULE_DESCRIPTION(description);
MODULE_VERSION(version_string);
MODULE_DEVICE_TABLE(table_info);
MODULE_ALIAS(alternate_name);

```

对于 USB、PCI 等设备驱动，通常会创建一个 MODULE_DEVICE_TABLE，表明该驱动模块所支持的设备，如代码清单 4.6 所示。

代码清单 4.6 驱动所支持的设备列表

```

1 /* 对应此驱动的设备表 */
2 static struct usb_device_id skel_table [] = {
3 { USB_DEVICE(USB_SKEL_VENDOR_ID,
4     USB_SKEL_PRODUCT_ID) },
5 { } /* 表结束 */
6 };
7
8 MODULE_DEVICE_TABLE (usb, skel_table);

```

此时，并不需要读者理解 MODULE_DEVICE_TABLE 的作用，后续相关章节会有详细介绍。

4.8 模块的使用计数

Linux 2.4 内核中，模块自身通过 MOD_INC_USE_COUNT、MOD_DEC_USE_COUNT 宏来



管理自己被使用的计数。

Linux 2.6 内核提供了模块计数管理接口 `try_module_get(&module)` 和 `module_put(&module)`，从而取代 Linux 2.4 内核中的模块使用计数管理宏。模块的使用计数一般不必由模块自身管理，而且模块计数管理还考虑了 SMP 与 PREEMPT 机制的影响。

```
int try_module_get(struct module *module);
```

该函数用于增加模块使用计数；若返回为 0，表示调用失败，希望使用的模块没有被加载或正在被卸载中。

```
void module_put(struct module *module);
```

该函数用于减少模块使用计数。

`try_module_get()` 与 `module_put()` 的引入与使用与 Linux 2.6 内核下的设备模型密切相关。Linux 2.6 内核为不同类型的设备定义了 `struct module *owner` 域，用来指向管理此设备的模块。当开始使用某个设备时，内核使用 `try_module_get(dev->owner)` 去增加管理此设备的 `owner` 模块的使用计数；当不再使用此设备时，内核使用 `module_put(dev->owner)` 减少对管理此设备的 `owner` 模块的使用计数。这样，当设备在使用时，管理此设备的模块将不能被卸载。只有当设备不再被使用时，模块才允许被卸载。

在 Linux 2.6 内核下，对于设备驱动工程师而言，很少需要亲自调用 `try_module_get()` 与 `module_put()`，因为此时开发人员所写的驱动通常为支持某具体设备的 `owner` 模块，对此设备 `owner` 模块的计数管理由内核里更底层的代码如总线驱动或是此类设备共用的核心模块来实现，从而简化了设备驱动开发。

4.9 模块的编译

我们可以为代码清单 4.1 的模板编写一个简单的 Makefile:

```
KVERS = $(shell uname -r)

# Kernel modules
obj-m += hello.o

# Specify flags for the module compilation.
#EXTRA_CFLAGS=-g -O0

build: kernel_modules

kernel_modules:
    make -C /lib/modules/$(KVERS)/build M=$(CURDIR) modules

clean:
    make -C /lib/modules/$(KVERS)/build M=$(CURDIR) clean
```

该 Makefile 文件应该与源代码 `hello.c` 位于同一目录，开启其中的 `EXTRA_CFLAGS=-g -O0` 可以得到包含调试信息的 `hello.ko` 模块。运行 `make` 命令得到的模块可直接在 PC 上运行。

如果一个模块包括多个 `.c` 文件（如 `file1.c`、`file2.c`），则应该以如下方式编写 Makefile:

```
obj-m := modulename.o  
modulename-objs := file1.o file2.o
```

4.10 使用模块绕开 GPL

对于企业自己编写的驱动等内核代码，如果不编译为模块则无法绕开 GPL，编译为模块后企业在产品中使用模块，则公司对外不再需要提供对应的源代码，为了使公司产品所使用的 Linux 操作系统支持模块，需要完成如下工作。

- 在内核编译时应该选上“可以加载模块”，嵌入式产品一般不需要动态卸载模块，所以“可以卸载模块”不用选。
- 将我们编译的内核模块.ko 文件应该放置在目标文件系统的相关目录中。
- 产品的文件系统中应该包含了支持新内核的 insmod、lsmod、rmmod 等工具，由于嵌入式产品中一般不需要建立模块间依赖关系，所以 modprobe 可以不要，一般也不需要卸载模块，所以 rmmod 也可以不要。
- 在使用中用户可使用 insmod 命令手动加载模块，如 insmod xxx.ko。
- 但是一般而言，产品在启动过程中应该加载模块，在嵌入式产品 Linux 的启动过程中，加载企业自己的模块的最简单的方法是修改启动过程的 rc 脚本，增加 insmod ../../xxx.ko 这样的命令。用 busybox 做出的文件系统，通常修改 etc/init.d/rcS 文件。

4.11 总结

本章主要讲解了 Linux 内核模块的概念和基本的编程方法。内核模块由加载/卸载函数、功能函数以及一系列声明组成，它可以被传入参数，也可以导出符号供其他模块使用。

由于 Linux 设备驱动以内核模块的形式而存在，因此，掌握这一章的内容是编写任何类型设备驱动的必须。在具体的设备驱动开发中，将驱动编译为模块也有很强的工程意义，因为如果将正在开发中的驱动直接编译入内核，而开发过程中会不断修改驱动的代码，则需要不断地编译内核并重启 Linux，但是如果编译为模块，则只需要 rmmod 并 insmod 即可，开发效率大为提高。

LINUX

第5章 Linux 文件系统与设备文件系统

本章导读

由于字符设备和块设备都良好地体现了“一切都是文件”的设计思想，掌握 Linux 文件系统、设备文件系统的知识就显得相当重要了。

首先，驱动工程师编写的驱动最终通过操作系统的文件操作系统调用或 C 库函数（本质也基于系统调用）被访问，而设备驱动的结构最终也是为了迎合提供给应用程序员的 API。

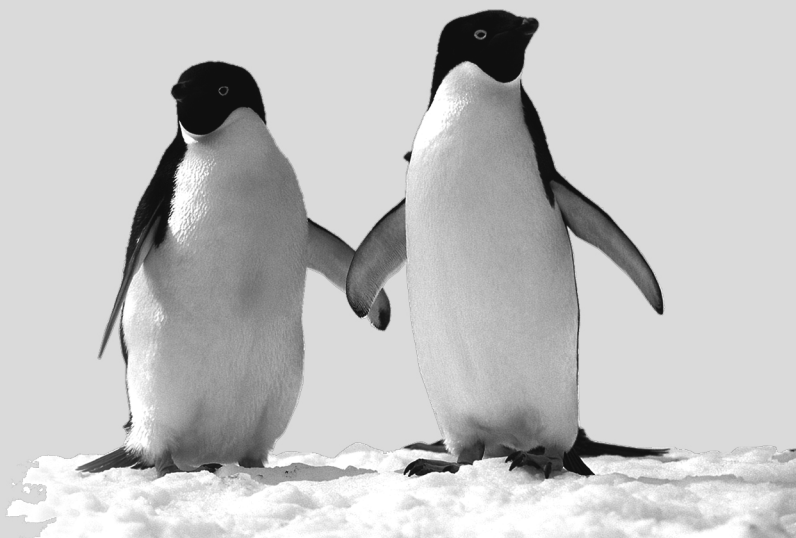
其次，驱动工程师在设备驱动中不可避免地会与设备文件系统打交道，从 Linux 2.4 内核的 devfs 文件系统到目前 Linux 2.6 基于 sysfs 的 udev 文件系统。

5.1 节讲解了通过 Linux API 和 C 库函数在用户空间进行 Linux 文件操作的编程方法。

5.2 节分析了 Linux 文件系统的目录结构，简单介绍了 Linux 内核中文件系统的实现，并给出了文件系统与设备驱动的关系。

5.3 节和 5.4 节分别讲解 Linux 2.4 内核的 devfs 和 Linux 2.6 所采用的 udev 设备文件系统，并分析了两者的区别。

5.5 节讲解了 LDD6410 的 SD 卡和 NAND 分区和文件系统的使用情况。



5.1 Linux 文件操作

5.1.1 文件操作系统调用

Linux 的文件操作系统调用（在 Windows 编程领域，习惯称操作系统提供的接口为 API）涉及创建、打开、读写和关闭文件。

1. 创建

```
int creat(const char *filename, mode_t mode);
```

参数 `mode` 指定新建文件的存取权限，它同 `umask` 一起决定文件的最终权限（`mode&umask`），其中 `umask` 代表了文件在创建时需要去掉的一些存取权限。`umask` 可通过系统调用 `umask()` 来改变：

```
int umask(int newmask);
```

该调用将 `umask` 设置为 `newmask`，然后返回旧的 `umask`，它只影响读、写和执行权限。

2. 打开

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

`open()` 函数有两个形式，其中 `pathname` 是我们要打开的文件名（包含路径名称，缺省是认为在当前路径下面），`flags` 可以是如表 5.1 所示的一个值或者是几个值的组合。

表 5.1 文件打开标志

标 志	含 义
O_RDONLY	以只读的方式打开文件
O_WRONLY	以只写的方式打开文件
O_RDWR	以读写的方式打开文件
O_APPEND	以追加的方式打开文件
O_CREAT	创建一个文件
O_EXEC	如果使用了 O_CREAT 而且文件已经存在，就会发生一个错误
O_NOBLOCK	以非阻塞的方式打开一个文件
O_TRUNC	如果文件已经存在，则删除文件的内容

O_RDONLY、O_WRONLY、O_RDWR 三个标志只能使用任意的一个。

如果使用了 O_CREATE 标志，则使用的函数是 `int open(const char *pathname,int flags,mode_t mode)`；这个时候我们还要指定 `mode` 标志，用来表示文件的访问权限。`mode` 可以是如表 5.2 所列值的组合。

表 5.2 文件访问权限

标 志	含 义
S_IRUSR	用户可以读
S_IWUSR	用户可以写
S_IXUSR	用户可以执行



续表

标 志	含 义
S_IRWXU	用户可以读、写、执行
S_IRGRP	组可以读
S_IWGRP	组可以写
S_IXGRP	组可以执行
S_IRWXG	组可以读、写、执行
S_IROTH	其他人可以读
S_IWOTH	其他人可以写
S_IXOTH	其他人可以执行
S_IRWXO	其他人可以读、写、执行
S_ISUID	设置用户执行 ID
S_ISGID	设置组的执行 ID

除了可以通过上述宏进行“或”逻辑产生标志以外,我们也可以自己用数字来表示,Linux 用 5 个数字来表示文件的各种权限:第一位表示设置用户 ID;第二位表示设置组 ID;第三位表示用户自己的权限位;第四位表示组的权限;最后一位表示其他人的权限。每个数字可以取 1(执行权限)、2(写权限)、4(读权限)、0(无)或者是这些值的和。例如,要创建一个用户可读、可写、可执行,但是组没有权限,其他人可以读、可以执行的文件,并设置用户 ID 位。那么,我们应该使用的模式是 1(设置用户 ID)、0(不设置组 ID)、7(1+2+4,读、写、执行)、0(没有权限)、5(1+4,读、执行)即 10 705:

```
open("test", O_CREAT, 10 705);
```

上述语句等价于:

```
open("test", O_CREAT, S_IRWXU | S_IROTH | S_IXOTH | S_ISUID );
```

如果文件打开成功,open 函数会返回一个文件描述符,以后对该文件的所有操作就可以通过对这个文件描述符进行操作来实现。

3. 读写

在文件打开以后,我们才可对文件进行读写,Linux 中提供文件读写的系统调用是 read、write 函数:

```
int read(int fd, const void *buf, size_t length);  
int write(int fd, const void *buf, size_t length);
```

其中参数 buf 为指向缓冲区的指针,length 为缓冲区的大小(以字节为单位)。函数 read()实现从文件描述符 fd 所指定的文件中读取 length 个字节到 buf 所指向的缓冲区中,返回值为实际读取的字节数。函数 write 实现将把 length 个字节从 buf 指向的缓冲区中写到文件描述符 fd 所指向的文件中,返回值为实际写入的字节数。

以 O_CREAT 为标志的 open 实际上实现了文件创建的功能,因此,下面的函数等同 creat()函数:

```
int open(pathname, O_CREAT | O_WRONLY | O_TRUNC, mode);
```

4. 定位

对于随机文件,我们可以随机地指定位置读写,使用如下函数进行定位:

```
int lseek(int fd, offset_t offset, int whence);
```

`lseek()`将文件读写指针相对 `whence` 移动 `offset` 个字节。操作成功时，返回文件指针相对于文件头的位置。参数 `whence` 可使用下述值：

`SEEK_SET`：相对文件开头

`SEEK_CUR`：相对文件读写指针的当前位置

`SEEK_END`：相对文件末尾

`offset` 可取负值，例如下述调用可将文件指针相对当前位置向前移动 5 个字节：

```
lseek(fd, -5, SEEK_CUR);
```

由于 `lseek` 函数的返回值为文件指针相对于文件头的位置，因此下列调用的返回值就是文件的长度：

```
lseek(fd, 0, SEEK_END);
```

5. 关闭

当我们操作完成以后，我们要关闭文件了，只要调用 `close` 就可以了，其中 `fd` 是我们关闭的文件描述符：

```
int close(int fd);
```

例程：编写一个程序，在当前目录下创建用户可读写文件 `hello.txt`，在其中写入 “Hello, software weekly”，关闭该文件。再次打开该文件，读取其中的内容并输出在屏幕上。

解答如代码清单 5.1。

代码清单 5.1 Linux 文件操作用户空间编程（使用系统调用）

```
1  #include <sys/types.h>
2  #include <sys/stat.h>
3  #include <fcntl.h>
4  #include <stdio.h>
5  #define LENGTH 100
6  main()
7  {
8      int fd, len;
9      char str[LENGTH];
10
11     fd = open("hello.txt", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR); /*
12     创建并打开文件 */
13     if (fd) {
14         write(fd, "Hello World", strlen("Hello World")); /*
15         写入字符串 */
16         close(fd);
17     }
18
19     fd = open("hello.txt", O_RDWR);
20     len = read(fd, str, LENGTH); /* 读取文件内容 */
21     str[len] = '\0';
22     printf("%s\n", str);
23     close(fd);
24 }
```

编译并运行，执行结果为输出 “Hello World”。

5.1.2 C 库文件操作

C 库函数的文件操作实际上是独立于具体的操作系统平台的，不管是在 DOS、Windows、Linux



还是在 VxWorks 中都是这些函数:

1. 创建和打开

```
FILE *fopen(const char *path, const char *mode);
```

fopen()实现打开指定文件 filename, 其中的 mode 为打开模式, C 库函数中支持的打开模式如表 5.3 所示。

表 5.3 C 库函数文件打开标志

标 志	含 义
r、rb	以只读方式打开
w、wb	以只写方式打开。如果文件不存在, 则创建该文件, 否则文件被截断
a、ab	以追加方式打开。如果文件不存在, 则创建该文件
r+、r+b、rb+	以读写方式打开
w+、w+b、wb+	以读写方式打开。如果文件不存在时, 创建新文件, 否则文件被截断
a+、a+b、ab+	以读和追加方式打开。如果文件不存在, 则创建新文件

其中 b 用于区分二进制文件和文本文件, 这一点在 DOS、Windows 系统中是有区分的, 但 Linux 不区分二进制文件和文本文件。

2. 读写

C 库函数支持以字符、字符串等单位, 支持按照某种格式进行文件的读写, 这一组函数为:

```
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
char *fgets(char *s, int n, FILE *stream);
int fputs(const char *s, FILE *stream);
int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t n, FILE *stream);
```

fread()实现从流 stream 中读取加 n 个字段, 每个字段为 size 字节, 并将读取的字段放入 ptr 所指的字符数组中, 返回实际已读取的字段数。在读取的字段数小于 num 时, 可能是在函数调用时出现错误, 也可能是读到文件的结尾。所以要通过调用 feof()和 ferror()来判断。

write()实现从缓冲区 ptr 所指的数组中把 n 个字段写到流 stream 中, 每个字段长为 size 个字节, 返回实际写入的字段数。

另外, C 库函数还提供了读写过程中的定位能力, 这些函数包括:

```
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, const fpos_t *pos);
int fseek(FILE *stream, long offset, int whence);
```

3. 关闭

利用 C 库函数关闭文件依然是很简单的操作:

```
int fclose(FILE *stream);
```

例程: 将第 5.1.1 节中的例程用 C 库函数来实现, 如代码清单 5-2 所示。

代码清单 5.2 Linux 文件操作用户空间编程 (使用 C 库函数)

```
1 #include <stdio.h>
2 #define LENGTH 100
```

```
3  main()
4  {
5      FILE *fd;
6      char str[LENGTH];
7
8      fd = fopen("hello.txt", "w+"); /* 创建并打开文件 */
9      if (fd) {
10         fputs("Hello World", fd); /* 写入字符串 */
11         fclose(fd);
12     }
13
14     fd = fopen("hello.txt", "r");
15     fgets(str, LENGTH, fd); /* 读取文件内容 */
16     printf("%s\n", str);
17     fclose(fd);
18 }
```

5.2 Linux 文件系统

5.2.1 Linux 文件系统目录结构

进入 Linux 根目录（即“/”，Linux 文件系统的入口，也是处于最高一级的目录），运行“ls -l”命令，看到 Linux 包含以下目录。

1. /bin

包含基本命令，如 ls、cp、mkdir 等，这个目录中的文件都是可执行的。

2. /sbin

包含系统命令，如 modprobe、hwclock、ifconfig 等，大多是涉及系统管理的命令，这个目录中的文件都是可执行的。

3. /dev

设备文件存储目录，应用程序通过对这些文件的读写和控制就可以访问实际的设备。

4. /etc

系统配置文件的所在地，一些服务器的配置文件也在这里，如用户账号及密码配置文件。busybox 的启动脚本也存放在该目录。

5. /lib

系统库文件存放目录，如 LDD6410 包含 libc-2.6.1.so、libpthread-2.6.1.so、libthread_db-1.0.so 等。

6. /mnt

/mnt 这个目录一般是用于存放挂载储存设备的挂载目录的，比如有 cdrom 等目录。可以参看/etc/fstab 的定义。有时我们可以把让系统开机自动挂载文件系统，把挂载点放在这里也是可以的。

7. /opt

opt 是“可选”的意思，有些软件包会被安装在这里，例如，在 LDD6410 的文件系统中，Qt/Embedded 就存放在该目录。



8. /proc

操作系统运行时, 进程及内核信息 (比如 CPU、硬盘分区、内存信息等) 存放在这里。/proc 目录为伪文件系统 proc 的挂载目录, proc 并不是真正的文件系统, 它存在于内存之中。

9. /tmp

有时用户运行程序的时候, 会产生临时文件, /tmp 就用来存放临时文件的。

10. /usr

这个是系统存放程序的目录, 比如用户命令、用户库等。LDD6410 的 usr 包括 bin、sbin、lib 三个子目录。usr/bin 中包含 diff、which、who、rx、cmp 等, usr/sbin 中包含 chroot、flash_eraseall、inetd 等, usr/lib 中包含 libjpeg.so.62.0.0 等。

11. /var

var 表示的是变化的意思, 这个目录的内容经常变动, 如/var 的/var/log 目录被用来存放系统日志。

12. /sys

Linux 2.6 内核所支持的 sysfs 文件系统被映射在此目录。Linux 设备驱动模型中的总线、驱动和设备都可以在 sysfs 文件系统中找到对应的节点。当内核检测到在系统中出现了新设备后, 内核会在 sysfs 文件系统中为该新设备生成一项新的记录。

5.2.2 Linux 文件系统与设备驱动

图 5.1 所示为 Linux 中虚拟文件系统、磁盘文件 (存放于 Ramdisk、Flash、ROM、SD 卡、U 盘等文件系统中的文件也属于此列) 及一般的设备文件与设备驱动程序之间的关系。

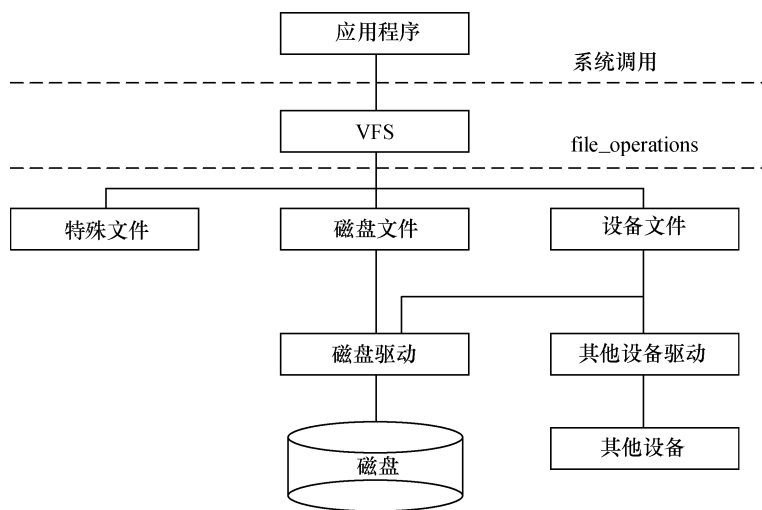


图 5.1 文件系统与设备驱动

应用程序和 VFS 之间的接口是系统调用, 而 VFS 与磁盘文件系统以及普通设备之间的接口是 file_operations 结构体成员函数, 这个结构体包含对文件进行打开、关闭、读写、控制的一系列成员函数。

由于字符设备的上层没有磁盘文件系统，所以字符设备的 `file_operations` 成员函数就直接由设备驱动提供了，在稍后的第 6 章，将会看到 `file_operations` 正是字符设备驱动的核心。

而对于块存储设备而言，`ext2`、`fat`、`jffs2` 等文件系统中会实现针对 VFS 的 `file_operations` 成员函数，设备驱动层将看不到 `file_operations` 的存在。磁盘文件系统和设备驱动会将对磁盘上文件的访问最终转换成对磁盘上柱面和扇区的访问。

在设备驱动程序的设计中，一般而言，会关心 `file` 和 `inode` 这两个结构体。

1. `file` 结构体

`file` 结构体代表一个打开的文件（设备对应于设备文件），系统中每个打开的文件在内核空间都有一个关联的 `struct file`。它由内核在打开文件时创建，并传递给在文件上进行操作的任何函数。在文件的所有实例都关闭后，内核释放这个数据结构。在内核和驱动源代码中，`struct file` 的指针通常被命名为 `file` 或 `filp`（即 `file pointer`）。代码清单 5.3 给出了文件结构体的定义。

代码清单 5.3 文件结构体

```
1 struct file
2 {
3     union {
4         struct list_head fu_list;
5         struct rcu_head fu_rcuhead;
6     } f_u;
7     struct dentry *f_dentry; /*与文件关联的目录入口(dentry)结构*/
8     struct vfsmount *f_vfsmnt;
9     struct file_operations *f_op; /* 和文件关联的操作*/
10    atomic_t f_count;
11    unsigned int f_flags; /*文件标志, 如 O_RDONLY、O_NONBLOCK、O_SYNC*/
12    mode_t f_mode; /*文件读/写模式, FMODE_READ 和 FMODE_WRITE*/
13    loff_t f_pos; /* 当前读写位置*/
14    struct fown_struct f_owner;
15    unsigned int f_uid, f_gid;
16    struct file_ra_state f_ra;
17
18    unsigned long f_version;
19    void *f_security;
20
21    /* tty 驱动需要, 其他的也许需要 */
22    void *private_data; /*文件私有数据*/
23    ...
24    struct address_space *f_mapping;
25 };
```

文件读/写模式 `mode`、标志 `f_flags` 都是设备驱动关心的内容，而私有数据指针 `private_data` 在设备驱动中被广泛应用，大多被指向设备驱动自定义用于描述设备的结构体。

驱动程序中经常会使用如下类似的代码来检测用户打开文件的读写方式：

```
if (file->f_mode & FMODE_WRITE) { /* 用户要求可写 */
    }
if (file->f_mode & FMODE_READ) { /* 用户要求可读 */
    }
}
```

下面的代码可用于判断以阻塞还是非阻塞方式打开设备文件：



```
if (file->f_flags & O_NONBLOCK)    /* 非阻塞 */
    pr_debug("open: non-blocking\n");
else                               /* 阻塞 */
    pr_debug("open: blocking\n");
```

2. inode 结构体

VFS inode 包含文件访问权限、属主、组、大小、生成时间、访问时间、最后修改时间等信息。它是 Linux 管理文件系统的最基本单位，也是文件系统连接任何子目录、文件的桥梁，inode 结构体的定义如代码清单 5.4 所示。

代码清单 5.4 inode 结构体

```
1 struct inode {
2     ...
3     umode_t i_mode; /* inode 的权限 */
4     uid_t i_uid; /* inode 拥有者的 id */
5     gid_t i_gid; /* inode 所属的群组 id */
6     dev_t i_rdev; /* 若是设备文件，此字段将记录设备的设备号 */
7     loff_t i_size; /* inode 所代表的文件大小 */
8
9     struct timespec i_atime; /* inode 最近一次的存取时间 */
10    struct timespec i_mtime; /* inode 最近一次的修改时间 */
11    struct timespec i_ctime; /* inode 的产生时间 */
12
13    unsigned long i_blksize; /* inode 在做 I/O 时的区块大小 */
14    unsigned long i_blocks; /* inode 所使用的 block 数，一个 block 为 512 byte */
15
16    struct block_device *i_bdev;
17    /*若是块设备，为其对应的 block_device 结构体指针*/
18    struct cdev *i_cdev; /*若是字符设备，为其对应的 cdev 结构体指针*/
19    ...
20};
```

对于表示设备文件的 inode 结构，i_rdev 字段包含设备编号。Linux 2.6 设备编号分为主设备编号和次设备编号，前者为 dev_t 的高 12 位，后者为 dev_t 的低 20 位。下列操作用于从一个 inode 中获得主设备号和次设备号：

```
unsigned int iminor(struct inode *inode);
unsigned int imajor(struct inode *inode);
```

查看 /proc/devices 文件可以获知系统中注册的设备，第 1 列为主设备号，第 2 列为设备名，如：

```
Character devices:
1 mem
2 pty
3 tty
4 /dev/vc/0
4 tty
5 /dev/tty
5 /dev/console
5 /dev/ptmx
7 vcs
10 misc
13 input
21 sg
29 fb
128 ptm
```



```
136 pts
171 ieee1394
180 usb
189 usb_device
```

```
Block devices:
 1 ramdisk
 2 fd
 8 sd
 9 md
22 idel
...
```

查看 `/dev` 目录可以获知系统中包含的设备文件，日期的前两列给出了对应设备的主设备号和次设备号：

```
crw-rw----  1 root    uucp      4,  64 Jan 30  2003 /dev/ttyS0
brw-rw----  1 root    disk     8,   0 Jan 30  2003 /dev/sda
```

主设备号是与驱动对应的概念，同一类设备一般使用相同的主设备号，不同类的设备一般使用不同的主设备号（但是也不排除在同一主设备号下包含有一定差异的设备）。因为同一驱动可支持多个同类设备，因此用次设备号来描述使用该驱动的设备的序号，序号一般从 0 开始。

内核 Documents 目录下的 `devices.txt` 文件描述了 Linux 设备号的分配情况，它由 LANANA（The Linux Assigned Names And Numbers Authority，网址：<http://www.lanana.org/>）组织维护，Torben Mathiasen（device@lanana.org）是其中的主要维护者。

5.3 devfs 设备文件系统

devfs（设备文件系统）是由 Linux 2.4 内核引入的，引入时被许多工程师给予了高度评价，它的出现使得设备驱动程序能自主地管理它自己的设备文件。具体来说，devfs 具有如下优点。

- （1）可以通过程序在设备初始化时在 `/dev` 目录下创建设备文件，卸载设备时将它删除。
- （2）设备驱动程序可以指定设备名、所有者和权限位，用户空间程序仍可以修改所有者和权限位。
- （3）不再需要为设备驱动程序分配主设备号以及处理次设备号，在程序中可以直接给 `register_chrdev()` 传递 0 主设备号以获得可用的主设备号，并在 `devfs_register()` 中指定次设备号。

驱动程序应调用下面这些函数来进行设备文件的创建和删除工作。

```
/*创建设备目录*/
devfs_handle_t devfs_mk_dir(devfs_handle_t dir, const char *name, void *info);
/*创建设备文件*/
devfs_handle_t devfs_register(devfs_handle_t dir, const char *name, unsigned
    int flags, unsigned int major, unsigned int minor, umode_t mode, void *ops,
    void *info);
/*撤销设备文件*/
void devfs_unregister(devfs_handle_t de);
```

在 Linux 2.4 的设备驱动编程中，分别在模块加载和卸载函数中创建和撤销设备文件是被普遍采用并值得大力推荐的好方法。代码清单 5.5 给出了一个使用 devfs 的例子。



代码清单 5.5 devfs 的使用范例

```
1 static devfs_handle_t devfs_handle;
2 static int __init xxx_init(void)
3 {
4     int ret;
5     int i;
6     /*在内核中注册设备*/
7     ret = register_chrdev(XXX_MAJOR, DEVICE_NAME, &xxx_fops);
8     if (ret < 0) {
9         printk(DEVICE_NAME " can't register major number\n");
10        return ret;
11    }
12    /*创建设备文件*/
13    devfs_handle =devfs_register(NULL, DEVICE_NAME, DEVFS_FL_DEFAULT,
14    XXX_MAJOR, 0, S_IFCHR | S_IRUSR | S_IWUSR, &xxx_fops, NULL);
15    ...
16    printk(DEVICE_NAME " initialized\n");
17    return 0;
18 }
19
20 static void __exit xxx_exit(void)
21 {
22     devfs_unregister(devfs_handle); /*撤销设备文件*/
23     unregister_chrdev(XXX_MAJOR, DEVICE_NAME); /*注销设备*/
24 }
25
26 module_init(xxx_init);
27 module_exit(xxx_exit);
```

代码中第 7 行和第 23 行分别用于注册和注销字符设备，使用的 `register_chrdev()` 和 `unregister_chrdev()` 在 Linux 2.6 内核中虽然仍然被支持，但这是过时的做法。第 13 和 22 行分别用于创建和删除 devfs 文件节点。

5.4 udev 设备文件系统

5.4.1 udev 与 devfs 的区别

尽管 devfs 有这样和那样的优点，但是，在 Linux 2.6 内核中，devfs 被认为是过时的方法，并最终被抛弃，udev 取代了它。Linux VFS 内核维护者 Al Viro 指出了几点 udev 取代 devfs 的原因：

- (1) devfs 所做的工作被确信可以在用户态来完成。
- (2) devfs 被加入内核之时，大家寄望它的质量可以迎头赶上。
- (3) devfs 被发现了一些可修复和无法修复的 bug。
- (4) 对于可修复的 bug，几个月前就已经被修复了，其维护者认为一切良好。
- (5) 对于后者，同样是相当长一段时间以来没有改观了。
- (6) devfs 的维护者和作者对它感到失望并且已经停止了对代码的维护工作。

Linux 内核的两位贡献者，Richard Gooch (devfs 的作者) 和 Greg Kroah-Hartman (sysfs 的主

要作者)就 devfs/udev 进行了激烈的争论:

Greg: Richard had stated that udev was a proper replacement for DevFS.

Richard: Well, that's news to me!

Greg: DevFS should be taken out because policy should exist in userspace and not in the kernel.

Richard: SysFS, developed in large part by Greg, also implemented policy in the kernel.

Greg: DevFS was broken and unfixable

Richard: No proof. Never say never...

这段有趣的争论可意译如下:

Greg: Richard 已经指出, udev 是 DevFS 恰当的替代品。

Richard: 哦, 是哪个 Richard 说的? 我怎么不知道。

Greg: DevFS 应该下课, 因为策略应该位于用户空间而不是内核空间。

Richard: 哦, 我听说, 相当大部分由 Greg 完成的 sysfs 也在内核中实现了策略。

Greg: devfs 很蹩脚, 也不稳定。

Richard: 呵呵, 没证据, 别那么武断……

在 Richard Gooch 和 Greg Kroah-Hartman 的争论中, Greg Kroah-Hartman 使用的理论依据就在于 policy (策略) 不能位于内核空间。Linux 设计中强调的一个基本观点是机制和策略的分离。机制是做某样事情的固定的步奏、方法, 而策略就是每一个步奏所采取的不同方式。机制是相对固定的, 而每个步奏采用的策略是不固定的。机制是稳定的, 而策略则是灵活的, 因此, 在 Linux 内核中, 不应该实现策略。Richard Gooch 认为, 属于策略的东西应该被移到用户空间。这就是为什么 devfs 位于内核空间, 而 udev 确要移到用户空间的原因。

下面举一个通俗的例子来理解 udev 设计的出发点。以谈恋爱为例, Greg Kroah-Hartman 认为, 可以让内核提供谈恋爱的机制, 但是不能在内核空间限制跟谁谈恋爱, 不能把谈恋爱的策略放在内核空间。因为恋爱是自由的, 用户应该可以在用户空间中实现“萝卜白菜, 各有所爱”的理想, 可以根据对方的外貌、籍贯、性格等自由选择。对应 devfs 而言, 第 1 个相亲的女孩被命名为/dev/girl0, 第 2 个相亲的女孩被命名为/dev/girl1, 依此类推。而在用户空间实现的 udev 则可以使得用户实现这样的自由: 不管你中意的女孩第几个来, 只要它与你定义的规则符合, 都命名为/dev/mygirl!

udev 完全在用户态工作, 利用设备加入或移除时内核所发送的热插拔事件 (hotplug event) 来工作。在热插拔时, 设备的详细信息会由内核输出到位于/sys 的 sysfs 文件系统。udev 的设备命名策略、权限控制和事件处理都是在用户态下完成的, 它利用 sysfs 中的信息来进行创建设备文件节点等工作。热插拔时输出到 sysfs 中的设备的详细信息就是相亲对象的资料 (外貌、年龄、性格、籍贯等), 设备命名策略等就是择偶标准。devfs 是个蹩脚的婚姻介绍所, 它直接指定了谁和谁谈恋爱, 而 udev 则聪明得多, 它只是把资料交给客户, 让客户根据这些资料去选择和谁谈恋爱。

由于 udev 根据系统中硬件设备的状态动态更新设备文件, 进行设备文件的创建和删除等, 因此, 在使用 udev 后, /dev 目录下就会只包含系统中真正存在的设备了。

devfs 与 udev 的另一个显著区别在于: 采用 devfs, 当一个并不存在的/dev 节点被打开的时候, devfs 能自动加载对应的驱动, 而 udev 则不这么做。这是因为 udev 的设计者认为 Linux 应该在设备被发现的时候加载驱动模块, 而不是当它被访问的时候。udev 的设计者认为 devfs 所



提供的打开/dev 节点时自动加载驱动的功能对于一个配置正确的计算机是多余的。系统中所有的设备都应该产生热插拔事件并加载恰当的驱动,而 udev 能注意到这点并且为它创建对应的设备节点。

5.4.2 sysfs 文件系统与 Linux 设备模型

Linux 2.6 的内核引入了 sysfs 文件系统,sysfs 被看成是与 proc、devfs 和 devpty 同类别的文件系统,该文件系统是一个虚拟的文件系统,它可以产生一个包括所有系统硬件的层级视图,与提供进程和状态信息的 proc 文件系统十分类似。

sysfs 把连接在系统上的设备和总线组织成为一个分级的文件,它们可以由用户空间存取,向用户空间导出内核数据结构以及它们的属性。sysfs 的一个目的就是展示设备驱动模型中各组件的层次关系,其顶级目录包括 block、device、bus、drivers、class、power 和 firmware。

block 目录包含所有的块设备;devices 目录包含系统所有的设备,并根据设备挂接的总线类型组织成层次结构;bus 目录包含系统中所有的总线类型;drivers 目录包括内核中所有已注册的设备驱动程序;class 目录包含系统中的设备类型(如网卡设备、声卡设备、输入设备等)。在/sys 目录运行 tree 会得到一个相当长的树型目录,下面摘取一部分:

```
|-- block
| | |-- fd0
| | |-- md0
| | |-- ram0
| | |-- ram1
| | |-- ...
|-- bus
| | |-- eisa
| | | |-- devices
| | | '-- drivers
| | |-- ide
| | |-- ieee1394
| | |-- pci
| | | |-- devices
| | | | |-- 0000:00:00.0 -> ../../../../devices/pci0000:00/0000:00:00.0
| | | | |-- 0000:00:01.0 -> ../../../../devices/pci0000:00/0000:00:01.0
| | | | |-- 0000:00:07.0 -> ../../../../devices/pci0000:00/0000:00:07.0
| | | '-- drivers
| | | | |-- PCI_IDE
| | | | | |-- bind
| | | | | |-- new_id
| | | | | '-- unbind
| | | | '-- pcnet32
| | | | | |-- 0000:00:11.0 -> ../../../../devices/pci0000:00/0000:00:11.0
| | | | | |-- bind
| | | | | |-- new_id
| | | | | '-- unbind
| | |-- platform
| | |-- pnp
| '-- usb
| | |-- devices
| | '-- drivers
|-- hub
```

```

|         |-- usb
|         |-- usb-storage
|         '-- usbfs
|-- class
| |-- graphics
| |-- hwmon
| |-- ieee1394
| |-- ieee1394_host
| |-- ieee1394_node
| |-- ieee1394_protocol
| |-- input
| |-- mem
| |-- misc
| |-- net
| |-- pci_bus
| | |-- 0000:00
| | | |-- bridge -> ../../../../devices/pci0000:00
| | | |-- cpuaffinity
| | | '-- uevent
| | '-- 0000:01
| | |-- bridge -> ../../../../devices/pci0000:00/0000:00:01.0
| | |-- cpuaffinity
| | '-- uevent
| |-- scsi_device
| |-- scsi_generic
| |-- scsi_host
| |-- tty
| |-- usb
| |-- usb_device
| |-- usb_host
| '-- vc
|-- devices
| |-- pci0000:00
| | |-- 0000:00:00.0
| | |-- 0000:00:07.0
| | |-- 0000:00:07.1
| | |-- 0000:00:07.2
| | |-- 0000:00:07.3
| |-- platform
| | |-- floppy.0
| | |-- host0
| | |-- i8042
| |-- pnp0
| '-- system
|-- firmware
|-- kernel
| '-- hotplug_seqnum
|-- module
| |-- apm
| |-- autofs
| |-- cdrom
| |-- eisa_bus
| |-- i8042
| '-- ide_cd

```



```
| |-- ide_scsi
| |-- ieee1394
| |-- md_mod
| |-- ohci1394
| |-- parport
| |-- parport_pc
| |-- usb_storage
| |-- usbcore
| | |-- parameters
| | |-- refcnt
| | '--- sections
| |-- virtual_root
| | '--- parameters
| |     '--- force_probe
| '--- vmhgfs
|     |-- refcnt
|     '--- sections
|         '--- __versions
'--- power
'--- state
```

在 `/sys/bus` 的 `pci` 等子目录下, 又会再分出 `drivers` 和 `devices` 目录, 而 `devices` 目录中的文件是对 `/sys/devices` 目录中文件的符号链接。同样地, `/sys/class` 目录下也包含许多对 `/sys/devices` 下文件的链接。如图 5.2 所示, 这与设备、驱动、总线和类的现实状况是直接对应的, 也正符合 Linux 2.6 的设备模型。

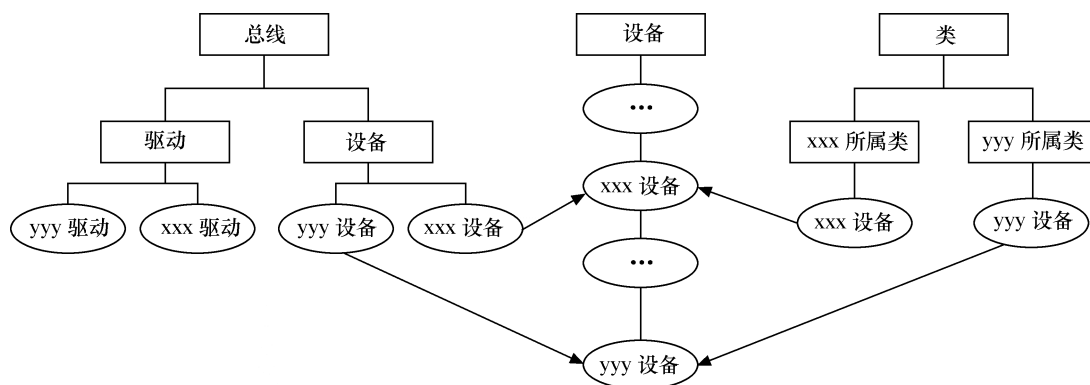


图 5.2 Linux 设备模型

随着技术的不断进步, 系统的拓扑结构越来越复杂, 对智能电源管理、热插拔以及即插即用的支持要求也越来越高, Linux 2.4 内核已经难以满足这些需求。为适应这种形势的需要, Linux 2.6 内核开发了上述全新的设备、总线、类和驱动环环相扣的设备模型。图 5.3 形象地表示了 Linux 驱动模型中设备、总线和类之间的关系。

大多数情况下, Linux 2.6 内核中的设备模型代码会作为“幕后黑手”处理好这些关系, 内核中的总线级和其他内核子系统会完成与设备模型的交互, 这使得驱动工程师几乎不需要关心设备模型。

在 Linux 内核中, 分别使用 `bus_type`、`device_driver` 和 `device` 来描述总线、驱动和设备, 这 3 个结构体定义于 `include/linux/device.h` 头文件中, 其定义如代码清单 5.6 所示。

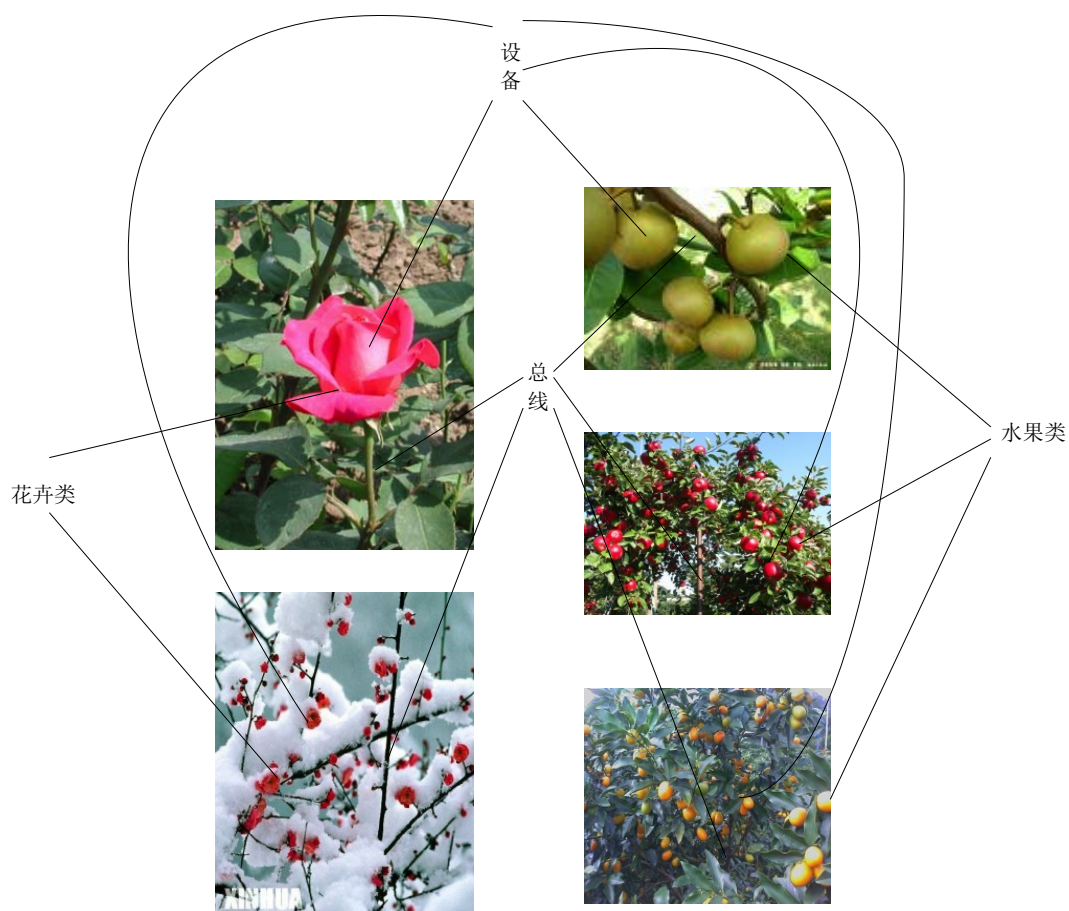


图 5.3 Linux 驱动模型中设备、总线和类的关系

代码清单 5.6 bus_type、device_driver 和 device 结构体

```

1 struct bus_type {
2     const char      *name;
3     struct bus_attribute *bus_attrs;
4     struct device_attribute *dev_attrs;
5     struct driver_attribute *drv_attrs;
6
7     int (*match)(struct device *dev, struct device_driver *drv);
8     int (*uevent)(struct device *dev, struct kobj_uevent_env *env);
9     int (*probe)(struct device *dev);
10    int (*remove)(struct device *dev);
11    void (*shutdown)(struct device *dev);
12
13    int (*suspend)(struct device *dev, pm_message_t state);
14    int (*suspend_late)(struct device *dev, pm_message_t state);
15    int (*resume_early)(struct device *dev);
16    int (*resume)(struct device *dev);
17
18    struct pm_ext_ops *pm;
19

```



```
20 struct bus_type_private *p;
21 };
22
23 struct device_driver {
24     const char      *name;
25     struct bus_type  *bus;
26
27     struct module    *owner;
28     const char      *mod_name;
29
30     int (*probe) (struct device *dev);
31     int (*remove) (struct device *dev);
32     void (*shutdown) (struct device *dev);
33     int (*suspend) (struct device *dev, pm_message_t state);
34     int (*resume) (struct device *dev);
35     struct attribute_group **groups;
36
37     struct pm_ops *pm;
38
39     struct driver_private *p;
40 };
41
42 struct device {
43     struct klist      klist_children;
44     struct klist_node knode_parent;
45     struct klist_node knode_driver;
46     struct klist_node knode_bus;
47     struct device     *parent;
48
49     struct kobject kobj;
50     char bus_id[BUS_ID_SIZE]; /* 在父总线中的位置 */
51     const char      *init_name; /* 设备的初始名 */
52     struct device_type *type;
53     unsigned        uevent_suppress:1;
54
55     struct semaphore sem;
56
57     struct bus_type *bus; /* 设备所在的总线类型 */
58     struct device_driver *driver; /* 设备用到的驱动 */
59     void *driver_data;
60     void *platform_data;
61     struct dev_pm_info power;
62
63 #ifdef CONFIG_NUMA
64     int numa_node;
65 #endif
66     u64 *dma_mask;
67     u64 coherent_dma_mask;
68
69     struct device_dma_parameters *dma_parms;
70
71     struct list_head dma_pools;
72
73     struct dma_coherent_mem *dma_mem;
74 }
```



```

75 struct dev_archdata archdata;
76
77 spinlock_t devres_lock;
78 struct list_head devres_head;
79
80 struct klist_node knode_class;
81 struct class *class;
82 dev_t devt; /* dev_t, 创建 sysfs "dev" */
83 struct attribute_group**groups;
84
85 void (*release)(struct device *dev);
86 };

```

`device_driver` 和 `device` 分别表示驱动和设备，而这两者都必须依附于一种总线，因此都包含 `struct bus_type` 指针。在 Linux 内核中，设备和驱动是分开注册的，注册 1 个设备的时候，并不需要驱动已经存在，而 1 个驱动被注册的时候，也不需要对应的设备已经被注册。设备和驱动各自涌向内核，而每个设备和驱动涌入的时候，都会去寻找自己的另一半。茫茫人海，何处觅踪？正是 `bus_type` 的 `match()` 成员函数将两者捆绑在一起。简单地说，设备和驱动就是红尘中漂浮的男女，而 `bus_type` 的 `match()` 则是牵引红线的月老，它可以识别什么设备与什么驱动可以配对。

注意，总线、驱动和设备都最终会落实为 `sysfs` 中的 1 个目录，因为进一步追踪代码会发现，它们实际上都可以认为是 `kobject` 的派生类（`device` 结构体直接包含了 `kobject kobj` 成员，而 `bus_type` 和 `device_driver` 则透过 `bus_type_private`、`driver_private` 间接包含 `kobject`），`kobject` 可看作所有总线、设备和驱动的抽象基类，1 个 `kobject` 对应 `sysfs` 中的 1 个目录。

总线、设备和驱动中的各个 `attribute` 则直接落实为 `sysfs` 中的 1 个文件，`attribute` 会伴随着 `show()` 和 `store()` 这两个函数，分别用于读和写该 `attribute` 对应的 `sysfs` 文件结点，代码清单 5.7 给出了 `attribute`、`bus_attribute`、`driver_attribute` 和 `device_attribute` 这几个结构体的定义。

代码清单 5.7 `attribute`、`bus_attribute`、`driver_attribute` 和 `device_attribute` 结构体

```

1 struct attribute {
2     const char *name;
3     struct module *owner;
4     mode_t mode;
5 };
6
7 struct bus_attribute {
8     struct attribute attr;
9     ssize_t (*show)(struct bus_type *bus, char *buf);
10    ssize_t (*store)(struct bus_type *bus, const char *buf, size_t count);
11 };
12
13 struct driver_attribute {
14     struct attribute attr;
15     ssize_t (*show)(struct device_driver *driver, char *buf);
16     ssize_t (*store)(struct device_driver *driver, const char *buf,
17                     size_t count);
18 };
19
20 struct device_attribute {
21     struct attribute attr;
22     ssize_t (*show)(struct device *dev, struct device_attribute *attr,

```



```
23         char *buf);
24     ssize_t (*store)(struct device *dev, struct device_attribute *attr,
25                     const char *buf, size_t count);
26 };
```

事实上, udev 规则中各信息的来源实际上就是 `bus_type`、`device_driver`、`device` 以及 `attribute` 等所对应 `sysfs` 节点。

5.4.3 udev 的组成

udev 的主页位于: <http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev.html>, 上面包含了关于 udev 的详细介绍, 从 <http://www.us.kernel.org/pub/linux/utils/kernel/hotplug/> 上可以下载最新的 udev 包。udev 的设计目标如下。

- 在用户空间中执行。
 - (1) 动态建立/删除设备文件。
 - (2) 允许每个人都不用关心主/次设备号。
 - (3) 提供 LSB 标准名称。
 - (4) 如果需要, 可提供固定的名称。

为了提供这些功能, udev 以 3 个分割的子计划发展: `namedev`、`libsysfs` 和 `udev`。`namedev` 为设备命名子系统, `libsysfs` 提供访问 `sysfs` 文件系统从中获取信息的标准接口, `udev` 提供 `/dev` 设备节点文件的动态创建和删除策略。`udev` 程序背负与 `namedev` 和 `libsysfs` 库交互的任务, 当 `/sbin/hotplug` 程序被内核调用时, `udev` 将被运行。`udev` 的工作过程如下。

(1) 当内核检测到在系统中出现了新设备后, 内核会在 `sysfs` 文件系统中为该新设备生成新的记录并导出一些设备特定的信息及所发生的事件。

(2) `udev` 获取内核导出的信息, 它调用 `namedev` 决定应该给该设备指定的名称, 如果是新插入设备, `udev` 将调用 `libsysfs` 决定应该为该设备的设备文件指定的主/次设备号, 并用分析获得的设备名称和主/次设备号创建 `/dev` 中的设备文件; 如果是设备移除, 则之前已经被创建的 `/dev` 文件将被删除。

在 `namedev` 中使用 5 步序列来决定指定设备的命名。

(1) 标签 (label) / 序号 (serial): 这一步检查设备是否有惟一的识别记号, 例如 USB 设备有惟一的 USB 序号, SCSI 有惟一的 UUID。如果 `namedev` 找到与这种惟一编号相对应的规则, 它将使用该规则提供的名称。

(2) 设备总线号: 这一步会检查总线设备编号, 对于不可热插拔的环境, 这一步足以辨别设备。例如, PCI 总线编号在系统的使用期间内很少变更。如果 `namedev` 找到相对应的规则, 规则中的名称就会被使用。

(3) 总线上的拓扑: 当设备在总线上的位置匹配用户指定的规则时, 就会使用该规则指定的名称。

(4) 替换名称: 当内核提供的名称匹配指定的替代字符串时, 就会使用替代字符串指定的名称。

(5) 内核提供的名称: 这一步“包罗万象”, 如果以前的几个步骤都没有被提供, 默认的内核将被指定给该设备。

代码清单 5.8 给出了一个 `namedev` 命名规则的例子, 第 2、4 行定义的是符合第 1 步的规则,

第 6、8 行定义的是符合第 2 步的规则，第 11、14 行定义的是符合第 3 步的规则，第 16 行定义的是符合第 4 步的规则。

代码清单 5.8 namedev 命名规则

```
1 # USB Epson printer to be called lp_epson
2 LABEL, BUS="usb", serial="HXOLL0012202323480", NAME="lp_epson"
3 # USB HP printer to be called lp_hp,
4 LABEL, BUS="usb", serial="W09090207101241330", NAME="lp_hp"
5 # sound card with PCI bus id 00:0b.0 to be the first sound card
6 NUMBER, BUS="pci", id="00:0b.0", NAME="dsp"
7 # sound card with PCI bus id 00:07.1 to be the second sound card
8 NUMBER, BUS="pci", id="00:07.1", NAME="dspl"
9 # USB mouse plugged into the third port of the first hub to be
10 # called mouse0
11 TOPOLOGY, BUS="usb", place="1.3", NAME="mouse0"
12 # USB tablet plugged into the second port of the second hub to be
13 # called mouse1
14 TOPOLOGY, BUS="usb", place="2.2", NAME="mouse1"
15 # ttyUSB1 should always be called visor
16 REPLACE, KERNEL="ttyUSB1", NAME="visor"
```

5.4.4 udev 规则文件

udev 的规则文件以行为单位，以“#”开头的行代表注释行。其余的每一行代表一个规则。每个规则分成一个或多个匹配和赋值部分。匹配部分用匹配专用的关键字来表示，相应的赋值部分用赋值专用的关键字来表示。匹配关键字包括：ACTION（行为）、KERNEL（匹配内核设备名）、BUS（匹配总线类型）、SYSFS（匹配从 sysfs 得到的信息，比如 label、vendor、USB 序列号）、SUBSYSTEM（匹配子系统名）等，赋值关键字包括：NAME（创建的设备文件名）、SYMLINK（符号创建链接名）、OWNER（设置设备的所有者）、GROUP（设置设备的组）、IMPORT（调用外部程序）等。

例如，如下规则：

```
SUBSYSTEM=="net", ACTION=="add", SYSFS{address}=="00:0d:87:f6:59:f3", IMPORT="/sbin/
rename_netiface %k eth0"
```

其中的“匹配”部分有 3 项，分别是 SUBSYSTEM、ACTION 和 SYSFS。而“赋值”部分有一项，是 IMPORT。这个规则的意思是：当系统中出现的新硬件属于 net 子系统范畴，系统对该硬件采取的动作是加入这个硬件，且这个硬件在 sysfs 文件系统上的“address”信息等于“00:0d:87:f6:59:f3”时，对这个硬件在 udev 层次施行的动作是调用外部程序/sbin/rename_netiface，并传递给该程序两个参数，一个是“%k”，代表内核对该新设备定义的名称，另一个是“eth0”。

通过一个简单的例子可以看出 udev 和 devfs 在命名方面的差异。如果系统中有两个 USB 打印机，一个可能被称为/dev/usb/lp0，另外一个便是/dev/usb/lp1。但是到底哪个文件对应哪个打印机是无法确定的，lp0、lp1 和实际的设备没有一一对应的关系，映射关系会因为设备发现的顺序，打印机本身关闭等原因而不确定。因此，理想的方式是两个打印机应该采用基于它们的序列号或者其他标识信息的办法来进行确定的映射，devfs 无法做到这一点，udev 却可以做到。使用如下规则：

```
BUS="usb", SYSFS{serial}="HXOLL0012202323480", NAME="lp_epson", SYMLINK="printers/
epson_stylus"
```



该规则中的匹配项目有 BUS 和 SYSFS, 赋值项目为 NAME 和 SYMLINK, 它意味着当一台 USB 打印机的序列号为 “HXOLL0012202323480” 时, 创建/dev/lp_epson 文件, 并同时创建一个符号链接/dev/printers/epson_styles。序列号为 “HXOLL0012202323480” 的 USB 打印机不管何时被插入, 对应的设备名都是/dev/lp_epson, 而 devfs 显然无法实现设备的这种固定命名。

udev 规则的写法非常灵活, 在匹配部分, 可以通过 “*”、“?”、[a~c]、[1~9]等 shell 通配符来灵活匹配多个项目。*类似于 shell 中的*通配符, 代替任意长度的任意字符串, ?代替一个字符, [x~y]是访问定义。此外, %k 就是 KERNEL, %n 则是设备的 KERNEL 序号 (如存储设备的分区号)。

可以借助 udev 中的 udevinfo 工具查找规则文件可以利用的信息, 如运行 “udevinfo -a -p/sys/block/sda” 命令将得到:

```
Udevinfo starts with the device specified by the devpath and then
walks up the chain of parent devices. It prints for every device
found, all possible attributes in the udev rules key format.
A rule to match, can be composed by the attributes of the device
and the attributes from one single parent device.

looking at device '/block/sda':
  KERNEL=="sda"
  SUBSYSTEM=="block"
  DRIVER==" "
  ATTR{stat}=="      1 689      3 169      85 746      24 000      2 017      2 095      32 896
47 292      0      23 188      71 292"
  ATTR{size}=="6 291 456"
  ATTR{removable}=="0"
  ATTR{range}=="16"
  ATTR{dev}=="8:0"

looking at parent device '/devices/platform/host0/target0:0:0/0:0:0:0':
  KERNELS=="0:0:0:0"
  SUBSYSTEMS=="scsi"
  DRIVERS=="sd"
  ATTRS{ioerr_cnt}=="0x5"
  ATTRS{iodone_cnt}=="0xe86"
  ATTRS{iorequest_cnt}=="0xe86"
  ATTRS{iocounterbits}=="32"
  ATTRS{timeout}=="30"
  ATTRS{state}=="running"
  ATTRS{rev}=="1.0 "
  ATTRS{model}=="VMware Virtual S"
  ATTRS{vendor}=="VMware, "
  ATTRS{scsi_level}=="3"
  ATTRS{type}=="0"
  ATTRS{queue_type}=="none"
  ATTRS{queue_depth}=="3"
  ATTRS{device_blocked}=="0"

looking at parent device '/devices/platform/host0/target0:0:0:':
  KERNELS=="target0:0:0"
  SUBSYSTEMS==" "
  DRIVERS==" "
```

```

looking at parent device '/devices/platform/host0':
    KERNELS=="host0"
    SUBSYSTEMS==" "
    DRIVERS==" "

looking at parent device '/devices/platform':
    KERNELS=="platform"
    SUBSYSTEMS==" "
    DRIVERS==" "

```

5.4.5 创建和配置 mdev

在嵌入式系统中，通常可以用 udev 的轻量级版本 mdev，mdev 集成于 busybox（本书配套 VirtualBox 虚拟机/home/lihacker/develop/svn/ldd6410-read-only/utls/busybox-1.15.1 目录）中。在 busybox 的源代码目录运行 make menuconfig，进入“Linux System Utilities”子选项，选中 mdev 相关项目，如图 5-4 所示。

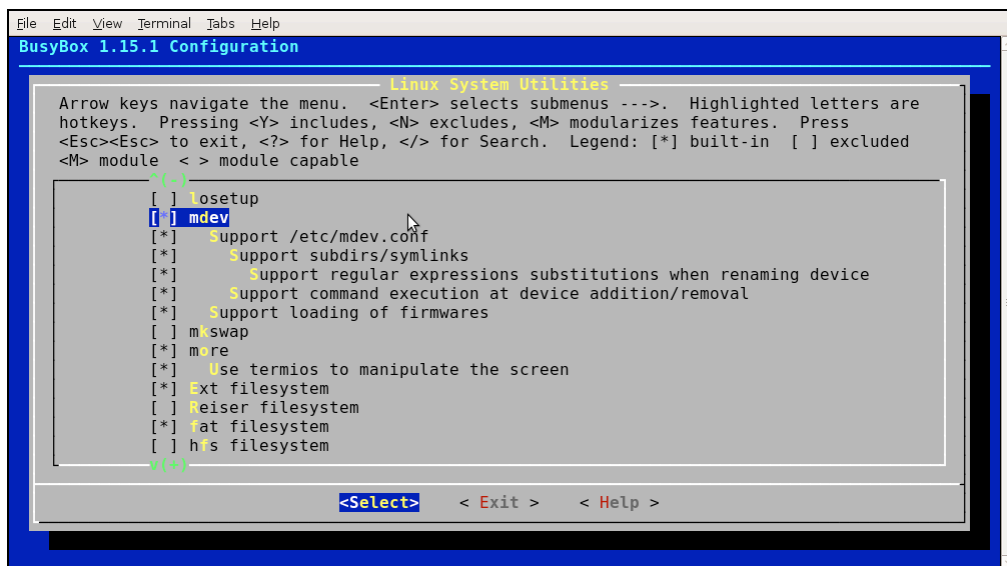


图 5.4 busybox 中的 mdev 选项

LDD6410 根文件系统中/etc/init.d/rcS 包含的如下内容即是为了使用 mdev 的功能：

```

/bin/mount -t sysfs sysfs /sys
/bin/mount -t tmpfs mdev /dev
echo /bin/mdev > /proc/sys/kernel/hotplug
mdev -s

```

其中“mdev -s”的含义是扫描/sys中所有的类设备目录，如果在目录中含有名为“dev”的文件，且文件中包含的是设备号，则 mdev 就利用这些信息为该设备在/dev下创建设备节点文件。

“echo /sbin/mdev > /proc/sys/kernel/hotplug”的含义是当有热插拔事件产生时，内核就会调用位于 /sbin 目录的 mdev。这时 mdev 通过环境变量中的 ACTION 和 DEVPATH，来确定此次热插拔事件的动作以及影响了/sys中的那个目录。接着会看看这个目录中是否有“dev”的属性文件，



如果有就利用这些信息为这个设备在/dev 下创建设备节点文件。

若要修改 mdev 的规则, 可通过修改/etc/mdev.conf 文件实现。

5.5 LDD6410 的 SD 和 NAND 文件系统

LDD6410 的 SD 卡分为两个区, 其中的第 2 个分区为 ext3 文件系统, 存放 LDD6410 的文件数据 (5.2.1 节给出的各个目录), 其制作方法如下。

(1) 在安装了 Linux 的 PC 机上通过 fdisk 给一张空的 SD 卡分为 2 个区 (如果 SD 卡中本身已经包含, 请通过 fdisk 的 “d” 命令全部删除), 得到如下的分区表:

```
Command (m for help): p

Disk /dev/sdb: 1 030 MB, 1 030 225 920 bytes
32 heads, 62 sectors/track, 1 014 cylinders
Units = cylinders of 1984 * 512 = 1 015 808 bytes
Disk identifier: 0x6f20736b

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1   *          1           20        19 809    83   Linux
/dev/sdb2             21         1014        98 6048    83   Linux
```

注意第 1 个分区制作的命令为:

```
Command (m for help): n
Command action
  e   extended
  p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-1 014, default 1):
Using default value 1
Last cylinder, +cylinders or +size{K,M,G} (1-1 014, default 1 014): 20M
```

第 2 个分区制作的命令是:

```
Command (m for help): n
Command action
  e   extended
  p   primary partition (1-4)
p
Partition number (1-4): 2
First cylinder (21-1 014, default 21):
Using default value 21
Last cylinder, +cylinders or +size{K,M,G} (21-1 014, default 1 014):
Using default value 1 014
```

```
Command (m for help):
```

我们还要通过 “a” 命令标记第 1 个分区:

```
Command (m for help): a
Partition number (1-4): 1
```

最后要通过 “w” 命令把建好的分区表写入 SD 卡。

(2) 格式化 SD 卡的分区 1 和分区 2:

```
mkfs.vfat /dev/sdb1
mkfs.ext3 /dev/sdb2
fsck.ext3 /dev/sdb2
```

(3) 如图 5-5 所示, 通过 moviNAND_Fusing_Tool.exe 烧写 SD 卡 U-BOOT 和 zImage。

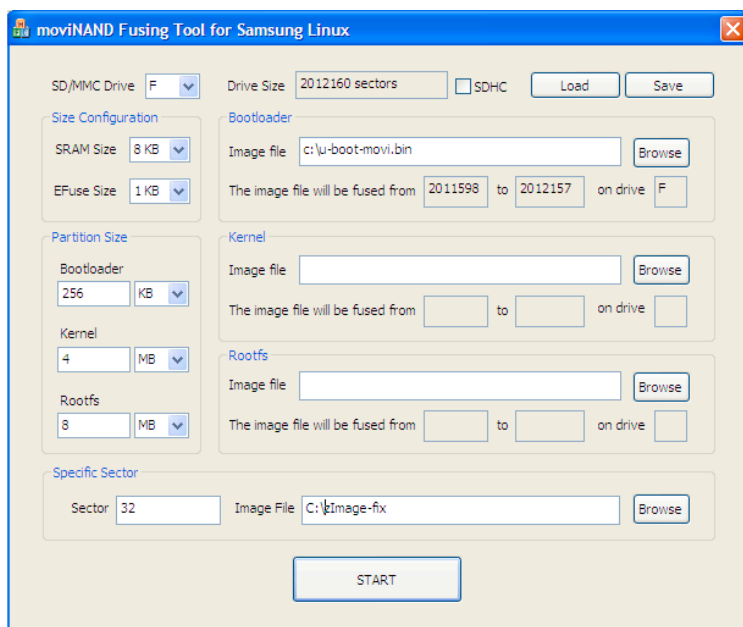


图 5.5 烧写 LDD6410 的 SD 卡 U-BOOT 和 zImage

更新 SD 卡根文件系统的方法很简单, 在 PC 机上 mount /dev/sdb2 后, 直接通过

```
cp -fa <your-rootfs> <sdb2-mount-point>
```

即可替换根文件系统了。<your-rootfs>是根文件系统的目录, <sdb2-mount-point>是/dev/sdb2 挂载的目录。

特别要注意的是, SD 的设备节点不一定是/dev/sdb, 应该视用户电脑的硬盘情况而言, 可能是/dev/sdc、/dev/sdd 等。

LDD6410 的 NAND 分为 3 个区, 分别存放 U-BOOT、zImage 和文件系统。该分区表定义在 LDD6410 的 BSP 中:

```
struct mtd_partition s3c_partition_info[] = {
    {
        .name      = "Bootloader",
        .offset    = 0,
        .size      = (512*SZ_1K),
    },
    {
        .name      = "Kernel",
        .offset    = (512*SZ_1K),
        .size      = (4*SZ_1M) - (512*SZ_1K),
    },
    {
        .name      = "File System",
```



```
        .offset      = MTDPART_OFS_APPEND,  
        .size        = MTDPART_SIZ_FULL,  
    }  
};
```

更新 NAND 中 U-BOOT 的方法如下:

(1) 通过 `tftp` 或 `nfs` 等方式获取新的 U-BOOT, 如:

```
# tftp -r u-boot-movi.bin -g 192.168.1.111
```

(2) 运行:

```
# flashcp u-boot-movi.bin /dev/mtd0
```

更新 NAND 中 zImage 的方法如下:

(1) 通过 `tftp` 或 `nfs` 等方式获取新的 zImage, 如:

```
# tftp -r zImage-fix -g 192.168.1.111
```

(2) 运行:

```
# flashcp zImage-fix /dev/mtd1
```

更新 NAND 中文档系统的方法如下:

在 PC 上将做好的新的根文件系统拷贝到 SD 卡或 NFS 的某目录, 下面我们以 `<new_rootfs_dir>` 指代该目录。

以 SD 卡或 NFS 为根文件系统启动系统, 运行如下命令擦除 `/dev/mtd2` 分区:

```
# flash_eraseall /dev/mtd2
```

然后将 NAND 的该分区 `mount` 到 `/mnt`:

```
# mount /dev/mtdblock2 -t yaffs2 /mnt/
```

将新的文件系统拷贝到 `/mnt`:

```
# cp -fa <new_rootfs_dir> /mnt
```

若上述命令运行过程中设备结点不存在, 可先执行:

```
# mdev -s
```

启动到 LDD6410, 在根目录下运行 `ls`, 会发现 LDD6410 根文件系统包含如下子目录:

```
#ls  
android          init.rc          sbin  
bin              lib             sqlite_stmt_journals  
cache            linuxrc         sys  
data             lost+found      system  
demo             mnt             tmp  
dev              opt             usr  
etc              proc            var  
init.goldfish.rc qtopia
```

其中的 `/data` `/cache` `/system` `/sqlite_stmt_journals` 为 Android 所需要的目录, `/opt` 下存放 Qt/Embedded 的可执行文件。运行 “`/android&`” 可启动 Android, 运行 “`/qtopia &`” 可启动 Qt/Embedded。

`/demo` 目录下存放在一些用于 demo 的图片、MP3 等, 如运行如下命令可显示 jpeg 图片 `1.jpg`、`2.jpg`、`3.jpg` 和 `4.jpg`。

```
# cd /demo/  
# jpegview 1.jpg 2.jpg 3.jpg 4.jpg  
480 272 480 544 0 272 16 0 480 272  
framebase = 0x4016c000 err=0  
ImageWidth=362 ImageHeight=272  
read 1.jpg OK  
ImageWidth=362 ImageHeight=272
```



```
read 2.jpg OK
ImageWidth=362 ImageHeight=272
read 3.jpg OK
ImageWidth=362 ImageHeight=272
read 4.jpg OK
```

5.6 总结

Linux 用户空间的文件编程有两种方法，即通过 Linux API 和通过 C 库函数访问文件。用户空间看不到设备驱动，能看到的只有设备对应的文件，因此文件编程即是用户空间的设备编程。

Linux 按照功能对文件系统的目录结构进行了良好的规划。`/dev` 是设备文件的存放目录，`devfs` 和 `udev` 分别是 Linux 2.4 和 Linux 2.6 生成设备文件节点的方法，前者运行于内核空间，后者运行于用户空间。

Linux 2.6 通过一系列数据结构定义了设备模型，设备模型与 `sysfs` 文件系统目录和文件存在一种对应关系，`udev` 可以利用 `sysfs` 中记录的信息定义规则并提取主次设备号动态创建 `/dev` 设备文件节点。

LINUX

第6章 字符设备驱动

本章导读

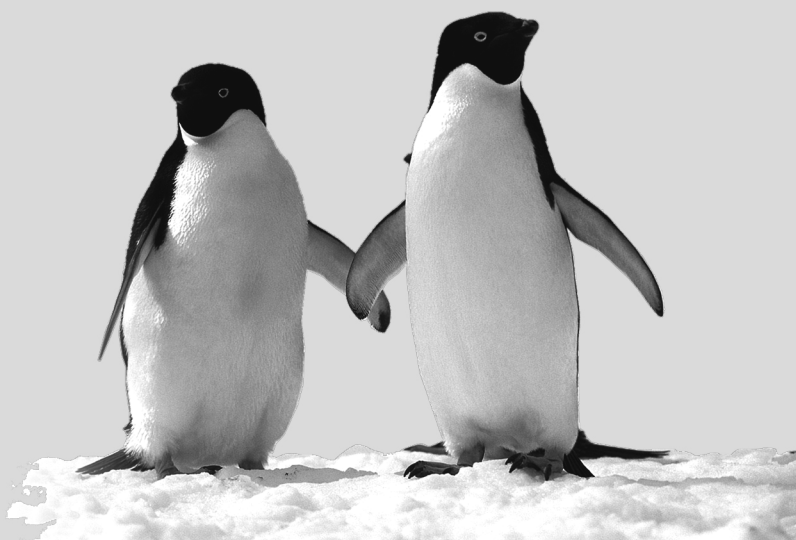
在整个 Linux 设备驱动的学习中，字符设备驱动较为基础。本章将讲解 Linux 字符设备驱动程序的结构，并解释其主要组成部分的编程方法。

6.1 节讲解了 Linux 字符设备驱动的关键数据结构 `cdev` 及 `file_operations` 结构体的操作方法，并分析了 Linux 字符设备的整体结构，给出了简单的设计模板。

6.2 节描述了本章及后续各章节所基于的 `globalmem` 虚拟字符设备，第 6~9 章都将基于该虚拟设备实例进行字符设备驱动及并发控制等知识的讲解。

6.3 节依据 6.1 节的知识讲解 `globalmem` 设备的驱动编写方法，对读写函数、`seek()`函数和 I/O 控制函数等进行了重点分析。该节的最后改造 `globalmem` 的驱动程序以利用文件私有数据。

6.4 节给出了 6.3 节的 `globalmem` 设备驱动在用户空间的验证。



6.1 Linux 字符设备驱动结构

6.1.1 cdev 结构体

在 Linux 2.6 内核中，使用 cdev 结构体描述一个字符设备，cdev 结构体的定义如代码清单 6.1。

代码清单 6.1 cdev 结构体

```
1 struct cdev {
2     struct kobject kobj; /* 内嵌的 kobject 对象 */
3     struct module *owner; /* 所属模块 */
4     struct file_operations *ops; /* 文件操作结构体 */
5     struct list_head list;
6     dev_t dev; /* 设备号 */
7     unsigned int count;
8 };
```

cdev 结构体的 dev_t 成员定义了设备号，为 32 位，其中 12 位主设备号，20 位次设备号。使用下列宏可以从 dev_t 获得主设备号和次设备号：

```
MAJOR(dev_t dev)
MINOR(dev_t dev)
```

而使用下列宏则可以通过主设备号和次设备号生成 dev_t：

```
MKDEV(int major, int minor)
```

cdev 结构体的另一个重要成员 file_operations 定义了字符设备驱动提供给虚拟文件系统的接口函数。

Linux 2.6 内核提供了一组函数用于操作 cdev 结构体：

```
void cdev_init(struct cdev *, struct file_operations *);
struct cdev *cdev_alloc(void);
void cdev_put(struct cdev *p);
int cdev_add(struct cdev *, dev_t, unsigned);
void cdev_del(struct cdev *);
```

cdev_init() 函数用于初始化 cdev 的成员，并建立 cdev 和 file_operations 之间的连接，其源代码如代码清单 6.2 所示。

代码清单 6.2 cdev_init() 函数

```
1 void cdev_init(struct cdev *cdev, struct file_operations *fops)
2 {
3     memset(cdev, 0, sizeof *cdev);
4     INIT_LIST_HEAD(&cdev->list);
5     kobject_init(&cdev->kobj, &ktype_cdev_default);
6     cdev->ops = fops; /* 将传入的文件操作结构体指针赋值给 cdev 的 ops */
7 }
```

cdev_alloc() 函数用于动态申请一个 cdev 内存，其源代码如代码清单 6.3 所示。

代码清单 6.3 cdev_alloc() 函数

```
1 struct cdev *cdev_alloc(void)
2 {
```



```
3     struct cdev *p = kzalloc(sizeof(struct cdev), GFP_KERNEL);
4     if (p) {
5         INIT_LIST_HEAD(&p->list);
6         kobject_init(&p->kobj, &ktype_cdev_dynamic);
7     }
8     return p;
9 }
```

cdev_add()函数和 cdev_del()函数分别向系统添加和删除一个 cdev, 完成字符设备的注册和注销。对 cdev_add()的调用通常发生在字符设备驱动模块加载函数中, 而对 cdev_del()函数的调用则通常发生在字符设备驱动模块卸载函数中。

6.1.2 分配和释放设备号

在调用 cdev_add()函数向系统注册字符设备之前, 应首先调用 register_chrdev_region()或 alloc_chrdev_region()函数向系统申请设备号, 这两个函数的原型为:

```
int register_chrdev_region(dev_t from, unsigned count, const char *name);
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count,
    const char *name);
```

register_chrdev_region()函数用于已知起始设备的设备号的情况, 而 alloc_chrdev_region()用于设备号未知, 向系统动态申请未被占用的设备号的情况, 函数调用成功之后, 会把得到的设备号放入第一个参数 dev 中。alloc_chrdev_region()与 register_chrdev_region()对比的优点在于它会自动避开设备号重复的冲突。

相反地, 在调用 cdev_del()函数从系统注销字符设备之后, unregister_chrdev_region()应该被调用以释放原先申请的设备号, 这个函数的原型为:

```
void unregister_chrdev_region(dev_t from, unsigned count);
```

6.1.3 file_operations 结构体

file_operations 结构体中的成员函数是字符设备驱动程序设计的主体内容, 这些函数实际会在应用程序进行 Linux 的 open()、write()、read()、close()等系统调用时最终被调用。file_operations 结构体目前已经比较庞大, 它的定义如代码清单 6.4 所示。

代码清单 6.4 file_operations 结构体

```
1 struct file_operations {
2     struct module *owner;
3     /* 拥有该结构的模块的指针, 一般为 THIS_MODULE */
4     loff_t(*llseek)(struct file *, loff_t, int);
5     /* 用来修改文件当前的读写位置 */
6     ssize_t(*read)(struct file *, char __user *, size_t, loff_t*);
7     /* 从设备中同步读取数据 */
8     ssize_t(*write)(struct file *, const char __user *, size_t, loff_t*);
9     /* 向设备发送数据 */
10    ssize_t(*aio_read)(struct kiocb *, char __user *, size_t, loff_t);
11    /* 初始化一个异步的读取操作 */
12    ssize_t(*aio_write)(struct kiocb *, const char __user *, size_t, loff_t);
13    /* 初始化一个异步的写入操作 */
14    int(*readdir)(struct file *, void *, filldir_t);
15    /* 仅用于读取目录, 对于设备文件, 该字段为 NULL */
16    unsigned int(*poll)(struct file *, struct poll_table_struct*);
```

```

17  /* 轮询函数, 判断目前是否可以进行非阻塞的读取或写入*/
18  int(*ioctl)(struct inode *, struct file *, unsigned int, unsigned long);
19  /* 执行设备 I/O 控制命令*/
20  long(*unlocked_ioctl)(struct file *, unsigned int, unsigned long);
21  /* 不使用 BLK 的文件系统, 将使用此种函数指针代替 ioctl */
22  long(*compat_ioctl)(struct file *, unsigned int, unsigned long);
23  /* 在 64 位系统上, 32 位的 ioctl 调用, 将使用此函数指针代替*/
24  int(*mmap)(struct file *, struct vm_area_struct*);
25  /* 用于请求将设备内存映射到进程地址空间*/
26  int(*open)(struct inode *, struct file*);
27  /* 打开 */
28  int(*flush)(struct file*);
29  int(*release)(struct inode *, struct file*);
30  /* 关闭*/
31  int (*fsync) (struct file *, struct dentry *, int datasync);
32  /* 刷新待处理的数据*/
33  int(*aio_fsync)(struct kiocb *, int datasync);
34  /* 异步 fsync */
35  int(*fasync)(int, struct file *, int);
36  /* 通知设备 FASYNC 标志发生变化*/
37  int(*lock)(struct file *, int, struct file_lock*);
38  ssize_t(*sendpage)(struct file *, struct page *, int, size_t, loff_t *, int);
39  /* 通常为 NULL */
40  unsigned long(*get_unmapped_area)(struct file *, unsigned long, unsigned long,
41  unsigned long, unsigned long);
42  /* 在当前进程地址空间找到一个未映射的内存段 */
43  int(*check_flags)(int);
44  /* 允许模块检查传递给 fcntl(F_SETTEL...)调用的标志 */
45  int(*dir_notify)(struct file *filp, unsigned long arg);
46  /* 对文件系统有效, 驱动程序不必实现*/
47  int(*flock)(struct file *, int, struct file_lock*);
48  ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t,
49  unsigned int); /* 由 VFS 调用, 将管道数据粘接到文件 */
50  ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t,
51  unsigned int); /* 由 VFS 调用, 将文件数据粘接到管道 */
52  int (*setlease)(struct file *, long, struct file_lock **);
53 };

```

下面我们对 `file_operations` 结构体中的主要成员进行分析。

`llseek()` 函数用来修改一个文件的当前读写位置, 并将新位置返回, 在出错时, 这个函数返回一个负值。

`read()` 函数用来从设备中读取数据, 成功时函数返回读取的字节数, 出错时返回一个负值。

`write()` 函数向设备发送数据, 成功时该函数返回写入的字节数。如果此函数未被实现, 当用户进行 `write()` 系统调用时, 将得到 `-EINVAL` 返回值。

`readdir()` 函数仅用于目录, 设备节点不需要实现它。

`ioctl()` 提供设备相关控制命令的实现 (既不是读操作也不是写操作), 当调用成功时, 返回给调用程序一个非负值。

`mmap()` 函数将设备内存映射到进程内存中, 如果设备驱动未实现此函数, 用户进行 `mmap()` 系统调用时将获得 `-ENODEV` 返回值。这个函数对于帧缓冲等设备特别有意义。

当用户空间调用 Linux API 函数 `open()` 打开设备文件时, 设备驱动的 `open()` 函数最终被调用。驱动程序可以不实现这个函数, 在这种情况下, 设备的打开操作永远成功。与 `open()` 函数对应的



是 `release()` 函数。

`poll()` 函数一般用于询问设备是否可被非阻塞地立即读写。当询问的条件未触发时，用户空间进行 `select()` 和 `poll()` 系统调用将引起进程的阻塞。

`aio_read()` 和 `aio_write()` 函数分别对与文件描述符对应的设备进行异步读、写操作。设备实现这两个函数后，用户空间可以对该设备文件描述符调用 `aio_read()`、`aio_write()` 等系统调用进行读写。

6.1.4 Linux 字符设备驱动的组成

在 Linux 中，字符设备驱动由如下几个部分组成。

1. 字符设备驱动模块加载与卸载函数

在字符设备驱动模块加载函数中应该实现设备号的申请和 `cdev` 的注册，而在卸载函数中应实现设备号的释放和 `cdev` 的注销。

工程师通常习惯为设备定义一个设备相关的结构体，其包含该设备所涉及的 `cdev`、私有数据及信号量等信息。常见的设备结构体、模块加载和卸载函数形式如代码清单 6.5 所示。

代码清单 6.5 字符设备驱动模块加载与卸载函数模板

```
1  /* 设备结构体
2  struct xxx_dev_t {
3      struct cdev cdev;
4      ...
5  } xxx_dev;
6  /* 设备驱动模块加载函数
7  static int __init xxx_init(void)
8  {
9      ...
10     cdev_init(&xxx_dev.cdev, &xxx_fops); /* 初始化 cdev */
11     xxx_dev.cdev.owner = THIS_MODULE;
12     /* 获取字符设备号*/
13     if (xxx_major) {
14         register_chrdev_region(xxx_dev_no, 1, DEV_NAME);
15     } else {
16         alloc_chrdev_region(&xxx_dev_no, 0, 1, DEV_NAME);
17     }
18
19     ret = cdev_add(&xxx_dev.cdev, xxx_dev_no, 1); /* 注册设备*/
20     ...
21 }
22 /*设备驱动模块卸载函数*/
23 static void __exit xxx_exit(void)
24 {
25     unregister_chrdev_region(xxx_dev_no, 1); /* 释放占用的设备号*/
26     cdev_del(&xxx_dev.cdev); /* 注销设备*/
27     ...
28 }
```

2. 字符设备驱动的 `file_operations` 结构体中成员函数

`file_operations` 结构体中成员函数是字符设备驱动与内核的接口，是用户空间对 Linux 进行系统调用最终的落实者。大多数字符设备驱动会实现 `read()`、`write()` 和 `ioctl()` 函数，常见的字符设备驱动的这 3 个函数的形式如代码清单 6.6 所示。

代码清单 6.6 字符设备驱动读、写、I/O 控制函数模板

```

1  /* 读设备*/
2  ssize_t xxx_read(struct file *filp, char __user *buf, size_t count,
3                  loff_t*f_pos)
4  {
5      ...
6      copy_to_user(buf, ..., ...);
7      ...
8  }
9  /* 写设备*/
10 ssize_t xxx_write(struct file *filp, const char __user *buf, size_t count,
11                  loff_t *f_pos)
12 {
13     ...
14     copy_from_user(..., buf, ...);
15     ...
16 }
17 /* ioctl 函数 */
18 int xxx_ioctl(struct inode *inode, struct file *filp, unsigned int cmd,
19               unsigned long arg)
20 {
21     ...
22     switch (cmd) {
23     case XXX_CMD1:
24         ...
25         break;
26     case XXX_CMD2:
27         ...
28         break;
29     default:
30         /* 不能支持的命令 */
31         return - ENOTTY;
32     }
33     return 0;
34 }

```

设备驱动的阅读函数中，`filp` 是文件结构体指针，`buf` 是用户空间内存的地址，该地址在内核空间不能直接读写，`count` 是要读的字节数，`f_pos` 是读的位置相对于文件开头的偏移。

设备驱动的写函数中，`filp` 是文件结构体指针，`buf` 是用户空间内存的地址，该地址在内核空间不能直接读写，`count` 是要写的字节数，`f_pos` 是写的位置相对于文件开头的偏移。

由于内核空间与用户空间的内存不能直接互访，因此借助了函数 `copy_from_user()` 完成用户空间到内核空间的拷贝，以及 `copy_to_user()` 完成内核空间到用户空间的拷贝，见代码第 6 行和第 14 行。

完成内核空间和用户空间内存拷贝的 `copy_from_user()` 和 `copy_to_user()` 的原型分别为：

```

unsigned long copy_from_user(void *to, const void __user *from, unsigned long count);
unsigned long copy_to_user(void __user *to, const void *from, unsigned long count);

```

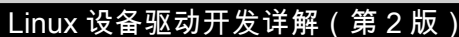
上述函数均返回不能被复制的字节数，因此，如果完全复制成功，返回值为 0。

如果要复制的内存是简单类型，如 `char`、`int`、`long` 等，则可以使用简单的 `put_user()` 和 `get_user()`，如：

```

int val; /* 内核空间整型变量
...
get_user(val, (int *) arg); /* 用户→内核，arg 是用户空间的地址

```



读和写函数中的 `_user` 是一个宏，表明其后的指针指向用户空间，这个宏定义为：

I/O 控制函数的 `cmd` 参数为事先定义的 I/O 控制命令，而 `arg` 为对应于该命令的参数。例如对于串行设备，如果 `SET_BAUDRATE` 是一道设置波特率的命令，那后面的 `arg` 就应该是波特率值。

在字符设备驱动中，需要定义一个 `file_operations` 的实例，并将具体设备驱动的函数赋值给 `file_operations` 的成员，如代码清单 6.7 所示。

代码清单 6.7 字符设备驱动文件操作结构体模板

上述 `xxx_fops` 在代码清单 6.5 第 10 行的 `cdev_init(&xxx_dev.cdev, &xxx_fops)` 的语句中被建立与 `cdev` 的连接。

图 6.1 所示为字符设备驱动的结构、字符设备驱动与字符设备以及字符设备驱动与用户空间访问该设备的程序之间的关系。

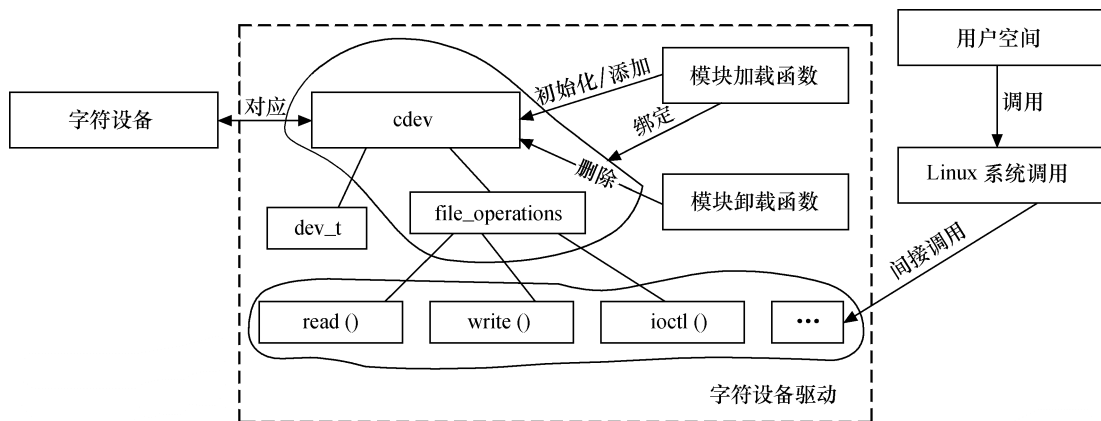


图 6.1 字符设备驱动的结构

6.2 globalmem 虚拟设备实例描述

从本章开始，后续的数字章都将基于虚拟的 globalmem 设备进行字符设备驱动的讲解。globalmem

意味着“全局内存”，在 `globalmem` 字符设备驱动中会分配一片大小为 `GLOBALMEM_SIZE` (4KB) 的内存空间，并在驱动中提供针对该片内存的读写、控制和定位函数，以供用户空间的进程能通过 Linux 系统调用访问这片内存。

实际上，这个虚拟的 `globalmem` 设备几乎没有任何实用价值，仅仅是一种为了讲解问题的方便而凭空制造的设备。当然，它也并非百无一用，由于 `globalmem` 可被两个或两个以上的进程同时访问，其中的全局内存可作为用户空间进程进行通信的一种蹩脚的手段。

本章将给出 `globalmem` 设备驱动的雏形，而后续章节会在这个雏形的基础上添加并发与同步控制等复杂功能。

6.3 globalmem 设备驱动

6.3.1 头文件、宏及设备结构体

在 `globalmem` 字符设备驱动中，应包含它要使用的头文件，并定义 `globalmem` 设备结构体及相关宏。

代码清单 6.8 `globalmem` 设备结构体和宏

```

1 #include <linux/module.h>
2 #include <linux/types.h>
3 #include <linux/fs.h>
4 #include <linux/errno.h>
5 #include <linux/mm.h>
6 #include <linux/sched.h>
7 #include <linux/init.h>
8 #include <linux/cdev.h>
9 #include <asm/io.h>
10 #include <asm/system.h>
11 #include <asm/uaccess.h>
12
13 #define GLOBALMEM_SIZE 0x1000 /*全局内存大小: 4KB*/
14 #define MEM_CLEAR 0x1 /*清零全局内存*/
15 #define GLOBALMEM_MAJOR 250 /*预设的 globalmem 的主设备号*/
16
17 static int globalmem_major = GLOBALMEM_MAJOR;
18 /*globalmem 设备结构体*/
19 struct globalmem_dev {
20     struct cdev cdev; /*cdev 结构体*/
21     unsigned char mem[GLOBALMEM_SIZE]; /*全局内存*/
22 };
23
24 struct globalmem_dev dev; /*设备结构体实例*/

```

从第 19~22 行代码可以看出，定义的 `globalmem_dev` 设备结构体包含了对应于 `globalmem` 字符设备的 `cdev`、使用的内存 `mem[GLOBALMEM_SIZE]`。当然，程序中并不一定要把 `mem[GLOBALMEM_SIZE]` 和 `cdev` 包含在一个设备结构体中，但这样定义的好处在于，它借用了面向对象程序设计中“封装”的思想，体现了一种良好的编程习惯。



6.3.2 加载与卸载设备驱动

globalmem 设备驱动的模块加载和卸载函数遵循代码清单 6.5 的类似模板, 其实现的工作与代码清单 6.5 完全一致, 如代码清单 6.9 所示。

代码清单 6.9 globalmem 设备驱动模块加载与卸载函数

```
1 /*globalmem 设备驱动模块加载函数*/
2 int globalmem_init(void)
3 {
4     int result;
5     dev_t devno = MKDEV(globalmem_major, 0);
6
7     /* 申请字符设备驱动区域*/
8     if (globalmem_major)
9         result = register_chrdev_region(devno, 1, "globalmem");
10    else {
11        /* 动态获得主设备号 */
12        result = alloc_chrdev_region(&devno, 0, 1, "globalmem");
13        globalmem_major = MAJOR(devno);
14    }
15    if (result < 0)
16        return result;
17
18    globalmem_setup_cdev();
19    return 0;
20 }
21
22 /*globalmem 设备驱动模块卸载函数*/
23 void globalmem_exit(void)
24 {
25     cdev_del(&dev.cdev); /*删除 cdev 结构*/
26     unregister_chrdev_region(MKDEV(globalmem_major, 0), 1);/*注销设备区域*/
27 }
```

第 18 行调用的 `globalmem_setup_cdev()` 函数完成 `cdev` 的初始化和添加, 如代码清单 6.10 所示。

代码清单 6.10 初始化并添加 `cdev` 结构体

```
1 /*初始化并添加 cdev 结构体*/
2 static void globalmem_setup_cdev()
3 {
4     int err, devno = MKDEV(globalmem_major, 0);
5
6     cdev_init(&dev.cdev, &globalmem_fops);
7     dev.cdev.owner = THIS_MODULE;
8     err = cdev_add(&dev.cdev, devno, 1);
9     if (err)
10         printk(KERN_NOTICE "Error %d adding globalmem", err);
11 }
```

在 `cdev_init()` 函数中, 与 `globalmem` 的 `cdev` 关联的 `file_operations` 结构体如代码清单 6.11 所示。

代码清单 6.11 globalmem 设备驱动文件操作结构体

```
1 static const struct file_operations globalmem_fops = {
2     .owner = THIS_MODULE,
```

```

3     .llseek = globalmem_llseek,
4     .read = globalmem_read,
5     .write = globalmem_write,
6     .ioctl = globalmem_ioctl,
7 };

```

6.3.3 读写函数

globalmem 设备驱动的读写函数主要是让设备结构体的 mem[] 数组与用户空间交互数据，并随着访问的字节数变更返回给用户的文件读写偏移位置。读和写函数的实现分别如代码清单 6.12 和 6.13 所示。

代码清单 6.12 globalmem 设备驱动读函数

```

1 static ssize_t globalmem_read(struct file *filp, char __user *buf, size_t count,
2     loff_t *ppos)
3 {
4     unsigned long p = *ppos;
5     int ret = 0;
6
7     /*分析和获取有效的读长度*/
8     if (p >= GLOBALMEM_SIZE) /* 要读的偏移位置越界
9         return 0;
10
11     if (count > GLOBALMEM_SIZE - p) /* 要读的字节数太大
12         count = GLOBALMEM_SIZE - p;
13
14     /*内核空间→用户空间*/
15     if (copy_to_user(buf, (void*)(dev.mem + p), count))
16         ret = -EFAULT;
17     else {
18         *ppos += count;
19         ret = count;
20
21         printk(KERN_INFO "read %d bytes(s) from %d\n", count, p);
22     }
23     return ret;
24 }

```

代码清单 6.13 globalmem 设备驱动写函数

```

1 static ssize_t globalmem_write(struct file *filp, const char __user *buf,
2     size_t count, loff_t *ppos)
3 {
4     unsigned long p = *ppos;
5     int ret = 0;
6
7     /*分析和获取有效的写长度*/
8     if (p >= GLOBALMEM_SIZE) /* 要写的偏移位置越界
9         return 0;
10
11     if (count > GLOBALMEM_SIZE - p) /* 要写的字节数太多
12         count = GLOBALMEM_SIZE - p;
13
14     /*用户空间→内核空间*/
15     if (copy_from_user(dev.mem + p, buf, count))

```



```
15     ret = - EFAULT;
16     else {
17         *ppos += count;
18         ret = count;
19
20         printk(KERN_INFO "written %d bytes(s) from %d\n", count, p);
21     }
22
23     return ret;
24 }
```

6.3.4 seek 函数

seek()函数对文件定位的起始地址可以是文件开头 (SEEK_SET, 0)、当前位置 (SEEK_CUR, 1) 和文件尾 (SEEK_END, 2), globalmem 支持从文件开头和当前位置相对偏移。

在定位的时候, 应该检查用户请求的合法性, 若不合法, 函数返回-EINVAL, 合法时返回文件的当前位置, 如代码清单 6.14。

代码清单 6.14 globalmem 设备驱动 seek()函数

```
1 static loff_t globalmem_llseek(struct file *filp, loff_t offset, int orig)
2 {
3     loff_t ret;
4     switch (orig) {
5
6     case 0: /*从文件开头开始偏移*/
7         if (offset < 0) {
8             ret = - EINVAL;
9             break;
10        }
11        if ((unsigned int)offset > GLOBALMEM_SIZE) {
12            ret = - EINVAL;
13            break;
14        }
15        filp->f_pos = (unsigned int)offset;
16        ret = filp->f_pos;
17        break;
18    case 1: /*从当前位置开始偏移*/
19        if ((filp->f_pos + offset) > GLOBALMEM_SIZE) {
20            ret = - EINVAL;
21            break;
22        }
23        if ((filp->f_pos + offset) < 0) {
24            ret = - EINVAL;
25            break;
26        }
27        filp->f_pos += offset;
28        ret = filp->f_pos;
29        break;
30    default:
31        ret = - EINVAL;
32    }
33    return ret;
34 }
```

6.3.5 ioctl 函数

1. globalmem 设备驱动的 ioctl()函数

globalmem 设备驱动的 ioctl()函数接受 MEM_CLEAR 命令，这个命令会将全局内存的有效数据长度清 0，对于设备不支持的命令，ioctl()函数应该返回- EINVAL，如代码清单 6.15 所示。

代码清单 6.15 globalmem 设备驱动的 I/O 控制函数

```
1 static int globalmem_ioctl(struct inode *inodep, struct file *filp, unsigned
2     int cmd, unsigned long arg)
3 {
4     switch (cmd) {
5     case MEM_CLEAR:
6         /* 清除全局内存
7          * memset(dev->mem, 0, GLOBALMEM_SIZE);
8          * printk(KERN_INFO "globalmem is set to zero\n");
9          * break;
10
11     default:
12         return - EINVAL; /* 其他不支持的命令
13     }
14     return 0;
15 }
```

在上述程序中，MEM_CLEAR 被宏定义为 0x01，实际上并不是一种值得推荐的方法，简单地对命令定义为 0x0、0x1、0x2 等类似值会导致不同的设备驱动拥有相同的命令号。如果设备 A、B 都支持 0x0、0x1、0x2 这样的命令，假设用户本身希望给 A 发 0x1 命令，可是不经意间发给了 B，这个时候 B 因为支持该命令，它就会执行该命令。因此，Linux 内核推荐采用一套统一的 ioctl()命令生成方式。

2. ioctl()命令

Linux 建议以如图 6.2 所示的方式定义 ioctl()的命令。

设备类型	序列号	方向	数据尺寸
8bit	8bit	2bit	13/14bit

图 6.2 I/O 控制命令的组成

命令码的设备类型字段为一个“幻数”，可以是 0~0xff 之间的值，内核中的 ioctl-number.txt 给出了一些推荐的和已经被使用的“幻数”，新设备驱动定义“幻数”的时候要避免与其冲突。

命令码的序列号也是 8 位宽。

命令码的方向字段为 2 位，该字段表示数据传送的方向，可能的值是_IOC_NONE（无数据传输）、_IOC_READ（读）、_IOC_WRITE（写）和_IOC_READ|_IOC_WRITE（双向）。数据传送的方向是从应用程序的角度来看的。

命令码的数据长度字段表示涉及的用户数据的大小，这个成员的宽度依赖于体系结构，通常是 13 或者 14 位。

内核还定义了_IOC()、_IOR()、_IOW()和_IOWR()这 4 个宏来辅助生成命令，这 4 个宏的通用



定义如代码清单 6.16 所示。

代码清单 6.16 _IO()、_IOR()、_IOW()和_IOWR()宏定义

```
1 #define _IO(type,nr) _IOC(_IOC_NONE, (type), (nr), 0)
2 #define _IOR(type,nr,size) _IOC(_IOC_READ, (type), (nr), \
3     (_IOC_TYPECHECK(size)))
4 #define _IOW(type,nr,size) _IOC(_IOC_WRITE, (type), (nr), \
5     (_IOC_TYPECHECK(size)))
6 #define _IOWR(type,nr,size) _IOC(_IOC_READ|_IOC_WRITE, (type), (nr), \
7     (_IOC_TYPECHECK(size)))
8 /* _IO、_IOR 等使用的 _IOC 宏 */
9 #define _IOC(dir,type,nr,size) \
10     (((dir) << _IOC_DIRSHIFT) | \
11     ((type) << _IOC_TYPSHIFT) | \
12     ((nr) << _IOC_NRSHIFT) | \
13     ((size) << _IOC_SIZESHIFT))
```

由此可见，这几个宏的作用是根据传入的 `type`（设备类型字段）、`nr`（序列号字段）和 `size`（数据长度字段）和宏名隐含的方向字段移位组合生成命令码。

由于 `globalmem` 的 `MEM_CLEAR` 命令不涉及数据传输，因此它可以定义为：

```
#define GLOBALMEM_MAGIC ...
#define MEM_CLEAR _IO(GLOBALMEM_MAGIC, 0)
```

3. 预定义命令

内核中预定义了一些 I/O 控制命令，如果某设备驱动中包含了与预定义命令一样的命令码，这些命令会被当作预定义命令被内核处理而不是被设备驱动处理，预定义命令有如下 4 种。

(1) `FIOCLEX`：即 File IOctl Close on Exec，对文件设置专用标志，通知内核当 `exec()` 系统调用发生时自动关闭打开的文件。

(2) `FIONCLEX`：即 File IOctl Not CClose on Exec，与 `FIOCLEX` 标志相反，清除由 `FIOCLEX` 命令设置的标志。

(3) `FIOQSIZE`：获得一个文件或者目录的大小，当用于设备文件时，返回一个 `ENOTTY` 错误。

(4) `FIONBIO`：即 File IOctl Non-Blocking I/O，这个调用修改在 `filp->f_flags` 中的 `O_NONBLOCK` 标志。

`FIOCLEX`、`FIONCLEX`、`FIOQSIZE` 和 `FIONBIO` 这些宏的定义为：

```
#define FIONCLEX 0x5450
#define FIOCLEX 0x5451
#define FIOQSIZE 0x5460
#define FIONBIO 0x5421
```

6.3.6 使用文件私有数据

6.3.1~6.3.5 节给出的代码完整地实现了预期的 `globalmem` 雏形，在其代码中，为 `globalmem` 设备结构体 `globalmem_dev` 定义了全局实例 `dev`（见代码清单 6.7 第 25 行），而 `globalmem` 的驱动中 `read()`、`write()`、`ioctl()`、`llseek()` 函数都针对 `dev` 进行操作。

实际上，大多数 Linux 驱动工程师遵循一个“潜规则”，那就是将文件的私有数据 `private_data` 指向设备结构体，在 `read()`、`write()`、`ioctl()`、`llseek()` 等函数通过 `private_data` 访问设备结构体。

这个时候，我们要将各函数进行少量的修改，为了让读者朋友建立字符设备驱动的全貌视

图，代码清单 6.17 列出了完整的使用文件私有数据的 globalmem 的设备驱动，本程序位于虚拟机/home/lihacker/develop/svn/ldd6410-read-only/training/kernel/drivers/globalmem/ch6 目录。

代码清单 6.17 使用文件私有数据的 globalmem 的设备驱动

```

1  #include <linux/module.h>
2  #include <linux/types.h>
3  #include <linux/fs.h>
4  #include <linux/errno.h>
5  #include <linux/mm.h>
6  #include <linux/sched.h>
7  #include <linux/init.h>
8  #include <linux/cdev.h>
9  #include <asm/io.h>
10 #include <asm/system.h>
11 #include <asm/uaccess.h>
12
13 #define GLOBALMEM_SIZE 0x1000 /*全局内存最大 4KB*/
14 #define MEM_CLEAR 0x1 /*清零全局内存*/
15 #define GLOBALMEM_MAJOR 250 /*预设的 globalmem 的主设备号*/
16
17 static int globalmem_major = GLOBALMEM_MAJOR;
18 /*globalmem 设备结构体*/
19 struct globalmem_dev {
20     struct cdev cdev; /*cdev 结构体*/
21     unsigned char mem[GLOBALMEM_SIZE]; /*全局内存*/
22 };
23
24 struct globalmem_dev *globalmem_devp; /*设备结构体指针*/
25 /*文件打开函数*/
26 int globalmem_open(struct inode *inode, struct file *filp)
27 {
28     /*将设备结构体指针赋值给文件私有数据指针*/
29     filp->private_data = globalmem_devp;
30     return 0;
31 }
32 /*文件释放函数*/
33 int globalmem_release(struct inode *inode, struct file *filp)
34 {
35     return 0;
36 }
37
38 /* ioctl 设备控制函数 */
39 static int globalmem_ioctl(struct inode *inodep, struct file *filp, unsigned
40     int cmd, unsigned long arg)
41 {
42     struct globalmem_dev *dev = filp->private_data; /*获得设备结构体指针*/
43
44     switch (cmd) {
45     case MEM_CLEAR:
46         memset(dev->mem, 0, GLOBALMEM_SIZE);
47         printk(KERN_INFO "globalmem is set to zero\n");
48         break;
49
50     default:

```



```
51         return -EINVAL;
52     }
53
54     return 0;
55 }
56
57 /*读函数*/
58 static ssize_t globalmem_read(struct file *filp, char __user *buf, size_t size,
59                             loff_t *ppos)
60 {
61     unsigned long p = *ppos;
62     unsigned int count = size;
63     int ret = 0;
64     struct globalmem_dev *dev = filp->private_data; /*获得设备结构体指针*/
65
66     /*分析和获取有效的写长度*/
67     if (p >= GLOBALMEM_SIZE)
68         return 0;
69     if (count > GLOBALMEM_SIZE - p)
70         count = GLOBALMEM_SIZE - p;
71
72     /*内核空间→用户空间*/
73     if (copy_to_user(buf, (void *) (dev->mem + p), count)) {
74         ret = -EFAULT;
75     } else {
76         *ppos += count;
77         ret = count;
78
79         printk(KERN_INFO "read %u bytes(s) from %lu\n", count, p);
80     }
81
82     return ret;
83 }
84
85 /*写函数*/
86 static ssize_t globalmem_write(struct file *filp, const char __user *buf,
87                               size_t size, loff_t *ppos)
88 {
89     unsigned long p = *ppos;
90     unsigned int count = size;
91     int ret = 0;
92     struct globalmem_dev *dev = filp->private_data; /*获得设备结构体指针*/
93
94     /*分析和获取有效的写长度*/
95     if (p >= GLOBALMEM_SIZE)
96         return 0;
97     if (count > GLOBALMEM_SIZE - p)
98         count = GLOBALMEM_SIZE - p;
99
100    /*用户空间→内核空间*/
101    if (copy_from_user(dev->mem + p, buf, count))
102        ret = -EFAULT;
103    else {
104        *ppos += count;
105        ret = count;
```



```

106
107     printk(KERN_INFO "written %u bytes(s) from %lu\n", count, p);
108 }
109
110     return ret;
111 }
112
113 /* seek 文件定位函数 */
114 static loff_t globalmem_llseek(struct file *filp, loff_t offset, int orig)
115 {
116     loff_t ret = 0;
117     switch (orig) {
118     case 0: /*相对文件开始位置偏移*/
119         if (offset < 0) {
120             ret = -EINVAL;
121             break;
122         }
123         if ((unsigned int)offset > GLOBALMEM_SIZE) {
124             ret = -EINVAL;
125             break;
126         }
127         filp->f_pos = (unsigned int)offset;
128         ret = filp->f_pos;
129         break;
130     case 1: /*相对文件当前位置偏移*/
131         if ((filp->f_pos + offset) > GLOBALMEM_SIZE) {
132             ret = -EINVAL;
133             break;
134         }
135         if ((filp->f_pos + offset) < 0) {
136             ret = -EINVAL;
137             break;
138         }
139         filp->f_pos += offset;
140         ret = filp->f_pos;
141         break;
142     default:
143         ret = -EINVAL;
144         break;
145     }
146     return ret;
147 }
148
149 /*文件操作结构体*/
150 static const struct file_operations globalmem_fops = {
151     .owner = THIS_MODULE,
152     .llseek = globalmem_llseek,
153     .read = globalmem_read,
154     .write = globalmem_write,
155     .ioctl = globalmem_ioctl,
156     .open = globalmem_open,
157     .release = globalmem_release,
158 };
159
160 /*初始化并注册 cdev*/

```



```
161 static void globalmem_setup_cdev(struct globalmem_dev *dev, int index)
162 {
163     int err, devno = MKDEV(globalmem_major, index);
164
165     cdev_init(&dev->cdev, &globalmem_fops);
166     dev->cdev.owner = THIS_MODULE;
167     err = cdev_add(&dev->cdev, devno, 1);
168     if (err)
169         printk(KERN_NOTICE "Error %d adding globalmem %d", err, index);
170 }
171
172 /*设备驱动模块加载函数*/
173 int globalmem_init(void)
174 {
175     int result;
176     dev_t devno = MKDEV(globalmem_major, 0);
177
178     /* 申请设备号*/
179     if (globalmem_major)
180         result = register_chrdev_region(devno, 1, "globalmem");
181     else { /* 动态申请设备号 */
182         result = alloc_chrdev_region(&devno, 0, 1, "globalmem");
183         globalmem_major = MAJOR(devno);
184     }
185     if (result < 0)
186         return result;
187
188     /* 动态申请设备结构体的内存*/
189     globalmem_devp = kmalloc(sizeof(struct globalmem_dev), GFP_KERNEL);
190     if (!globalmem_devp) { /*申请失败*/
191         result = - ENOMEM;
192         goto fail_malloc;
193     }
194
195     memset(globalmem_devp, 0, sizeof(struct globalmem_dev));
196
197     globalmem_setup_cdev(globalmem_devp, 0);
198     return 0;
199
200 fail_malloc:
201     unregister_chrdev_region(devno, 1);
202     return result;
203 }
204
205 /*模块卸载函数*/
206 void globalmem_exit(void)
207 {
208     cdev_del(&globalmem_devp->cdev); /*注销 cdev*/
209     kfree(globalmem_devp); /*释放设备结构体内存*/
210     unregister_chrdev_region(MKDEV(globalmem_major, 0), 1); /*释放设备号*/
211 }
212
213 MODULE_AUTHOR("Barry Song <21cnbao@gmail.com>");
214 MODULE_LICENSE("Dual BSD/GPL");
215
```

```

216 module_param(globalmem_major, int, S_IRUGO);
217
218 module_init(globalmem_init);
219 module_exit(globalmem_exit);

```

除了在 `globalmem_open()` 函数中通过 `filp->private_data = globalmem_devp` 语句（见第 29 行）将设备结构体指针赋值给文件私有数据指针并在 `globalmem_read()`、`globalmem_write()`、`globalmem_llseek()` 和 `globalmem_ioctl()` 函数中通过 `struct globalmem_dev *dev = filp->private_data` 语句获得设备结构体指针并使用该指针操作设备结构体外，代码清单 6.17 与代码清单 6.7~6.15 的程序基本相同。



读者朋友们，这个时候，请您翻回到本书的第 1 章，再次阅读代码清单 1.4，即 Linux 下 LED 的设备驱动，是否豁然开朗？

代码清单 6.17 仅仅作为使用 `private_data` 的范例，实际上，在这个程序中使用 `private_data` 没有任何意义，直接访问全局变量 `globalmem_devp` 会更加结构清晰。如果 `globalmem` 不只包括一个设备，而是同时包括两个或两个以上的设备，采用 `private_data` 的优势就会集中显现出来。

在不对代码清单 6.17 中的 `globalmem_read()`、`globalmem_write()`、`globalmem_ioctl()` 等重要函数及 `globalmem_fops` 结构体等数据结构进行任何修改的前提下，只是简单地修改 `globalmem_init()`、`globalmem_exit()` 和 `globalmem_open()`，就可以轻松地让 `globalmem` 驱动中包含两个同样的设备（次设备号分别为 0 和 1），如代码清单 6.18 所示。

代码清单 6.18 支持 2 个 `globalmem` 设备的 `globalmem` 驱动

```

1  /*文件打开函数*/
2  int globalmem_open(struct inode *inode, struct file *filp)
3  {
4      /*将设备结构体指针赋值给文件私有数据指针*/
5      struct globalmem_dev *dev;
6
7      dev = container_of(inode->i_cdev, struct globalmem_dev, cdev);
8      filp->private_data = dev;
9      return 0;
10 }
11
12 /*设备驱动模块加载函数*/
13 int globalmem_init(void)
14 {
15     int result;
16     dev_t devno = MKDEV(globalmem_major, 0);
17
18     /* 申请设备号*/
19     if (globalmem_major)
20         result = register_chrdev_region(devno, 2, "globalmem");
21     else { /* 动态申请设备号 */
22         result = alloc_chrdev_region(&devno, 0, 2, "globalmem");
23         globalmem_major = MAJOR(devno);
24     }
25
26     if (result < 0)

```



```
27     return result;
28
29     /* 动态申请两个设备结构体的内存*/
30     globalmem_devp = kmalloc(2*sizeof(struct globalmem_dev), GFP_KERNEL);
31     if (!globalmem_devp) { /*申请失败*/
32         result = - ENOMEM;
33         goto fail_malloc;
34     }
35     memset(globalmem_devp, 0, 2*sizeof(struct globalmem_dev));
36
37     globalmem_setup_cdev(&globalmem_devp[0], 0);
38     globalmem_setup_cdev(&globalmem_devp[1], 1);
39     return 0;
40
41 fail_malloc: unregister_chrdev_region(devno, 1);
42     return result;
43 }
44
45 /*模块卸载函数*/
46 void globalmem_exit(void)
47 {
48     cdev_del(&(globalmem_devp[0].cdev));
49     cdev_del(&(globalmem_devp[1].cdev)); /* 注销 cdev */
50     kfree(globalmem_devp); /*释放设备结构体内存*/
51     unregister_chrdev_region(MKDEV(globalmem_major, 0), 2); /*释放设备号*/
52 }
53
54 /* 其他代码同清单 6.16 */
```

代码清单 6.18 第 7 行调用的 `container_of()` 的作用是通过结构体成员的指针找到对应结构体的指针, 这个技巧在 Linux 内核编程中十分常用。在 `container_of(inode->i_cdev, struct globalmem_dev, cdev)` 语句中, 传给 `container_of()` 的第 1 个参数是结构体成员的指针, 第 2 个参数为整个结构体的类型, 第 3 个参数为传入的第 1 个参数即结构体成员的类型, `container_of()` 返回值为整个结构体的指针。

6.4 globalmem 驱动在用户空间的验证

在对应目录通过 “make” 命令编译 globalmem 的驱动, 得到 globalmem.ko 文件。运行:

```
lihacker@lihacker-laptop: ~ /develop/svn/ldd6410-read-only/training/kernel/drivers/globalmem/ch6$ sudo su
```

```
root@lihacker-laptop: /home/lihacker/develop/svn/ldd6410-read-only/training/kernel/drivers/globalmem/ch6# insmod globalmem.ko
```

命令加载模块, 通过 “lsmod” 命令, 发现 globalmem 模块已被加载。再通过 “cat /proc/devices” 命令查看, 发现多出了主设备号为 250 的 “globalmem” 字符设备驱动:

```
root@lihacker-laptop: /home/lihacker/develop/svn/ldd6410-read-only/training/kernel/drivers/globalmem/ch6# cat /proc/devices
Character devices:
1 mem
4 /dev/vc/0
```

```

4 tty
4 ttyS
5 /dev/tty
5 /dev/console
5 /dev/ptmx
6 lp
7 vcs
10 misc
13 input
14 sound
21 sg
29 fb
99 ppdev
108 ppp
116 alsa
128 ptm
136 pts
180 usb
188 ttyUSB
189 usb_device
216 rfcomm
226 drm
250 globalmem

```

接下来，通过命令：

```

root@lihacker-laptop:/home/lihacker/develop/svn/ldd6410-read-only/training/kernel/d
rivers/globalmem/ch6# mknod /dev/globalmem c 250 0

```

创建“/dev/globalmem”设备节点，并通过“echo 'hello world' > /dev/globalmem”命令和“cat /dev/globalmem”命令分别验证设备的写和读，结果证明“hello world”字符串被正确地写入 globalmem 字符设备：

```

root@lihacker-laptop:/home/lihacker/develop/svn/ldd6410-read-only/training/kernel/d
rivers/globalmem/ch6# echo "hello world" > /dev/globalmem

root@lihacker-laptop:/home/lihacker/develop/svn/ldd6410-read-only/training/kernel/d
rivers/globalmem/ch6# cat /dev/globalmem
hello world

```

如果启用了 sysfs 文件系统，将发现多出了 /sys/module/globalmem 目录，该目录下的树型结构为：

```

|-- refcnt
'-- sections
    |-- .bss
    |-- .data
    |-- .gnu.linkonce.this_module
    |-- .rodata
    |-- .rodata.str1.1
    |-- .strtab
    |-- .symtab
    |-- .text
    '-- __versions

```

refcnt 记录了 globalmem 模块的引用计数，sections 下包含的数个文件则给出了 globalmem 所包含的 BSS、数据段和代码段等的地址及其他信息。

对于代码清单 6.18 给出的支持两个 globalmem 设备的驱动，在加载模块后需创建两个设备节点，/dev/globalmem0 对应主设备号 globalmem_major，次设备号 0，/dev/globalmem1 对应主设备



号 `globalmem_major`，次设备号 1。分别读写 `/dev/globalmem0` 和 `/dev/globalmem1`，发现都读写到了正确的对应的设备。

6.5 总结

字符设备是 3 大类设备（字符设备、块设备和网络设备）中较简单的一类设备，其驱动程序中完成的主要工作是初始化、添加和删除 `cdev` 结构体，申请和释放设备号，以及填充 `file_operations` 结构体中的操作函数，实现 `file_operations` 结构体中的 `read()`、`write()` 和 `ioctl()` 等函数是驱动设计的主体工作。

LINUX

第7章

Linux 设备驱动中的并发控制

本章导读

Linux 设备驱动中必须解决的一个问题是多个进程对共享资源的并发访问，并发的访问会导致竞态，即使是经验丰富的驱动工程师也常常设计出包含并发问题 bug 的驱动程序。

Linux 提供了多种解决竞态问题的方式，这些方式适合不同的应用场景。7.1 节讲解了并发和竞态的概念及发生场合。7.2~7.5 节分别讲解了中断屏蔽、原子操作、自旋锁和信号量等并发控制机制。7.6 节讲解增加并发控制后的 globalmem 的设备驱动。





7.1 并发与竞态

并发 (concurrency) 指的是多个执行单元同时、并行被执行, 而并发的执行单元对共享资源 (硬件资源和软件上的全局变量、静态变量等) 的访问则很容易导致竞态 (race conditions)。例如, 对于 globalmem 设备, 假设一个执行单元 A 对其写入 3 000 个字符 “a”, 而另一个执行单元 B 对其写入 4 000 个 “b”, 第三个执行单元 C 读取 globalmem 的所有字符。如果执行单元 A、B 的写操作如图 7.1 那样顺序发生, 执行单元 C 的读操作当然不会有什么问题。但是, 如果执行单元 A、B 如图 7.2 那样被执行, 而执行单元 C 又 “不合时宜” 地读, 则会读出 3 000 个 “b”。

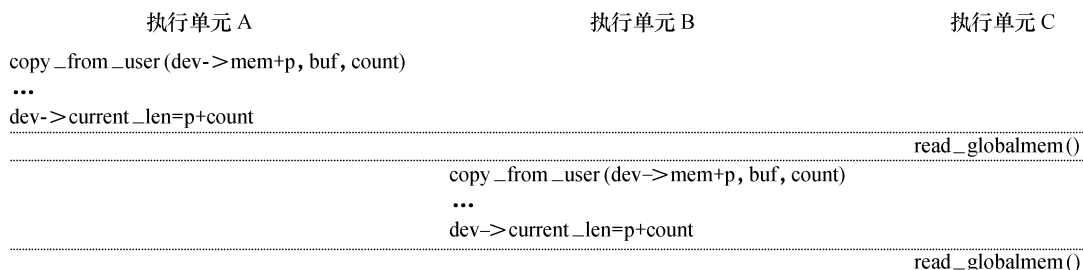


图 7.1 并发执行单元的顺序执行

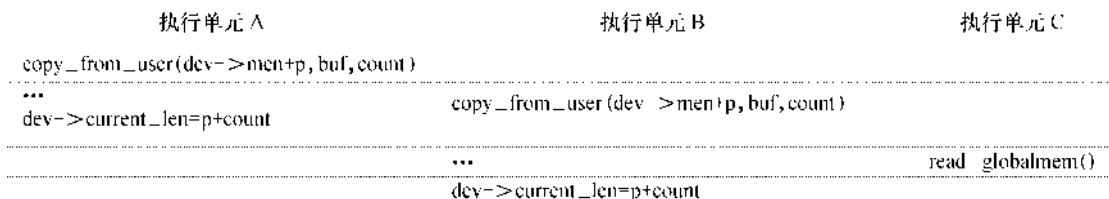


图 7.2 并发执行单元的交错执行

比图 7.2 更复杂、更混乱的并发大量地存在于设备驱动中, 只要并发的多个执行单元存在对共享资源的访问, 竞态就可能发生。在 Linux 内核中, 主要的竞态发生于如下几种情况。

1. 对称多处理器 (SMP) 的多个 CPU

SMP 是一种紧耦合、共享存储的系统模型, 其体系结构如图 7.3 所示, 它的特点是多个 CPU 使用共同的系统总线, 因此可访问共同的外设和存储器。

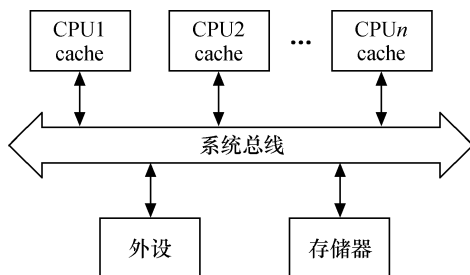


图 7.3 SMP 体系结构

2. 单 CPU 内进程与抢占它的进程

Linux 2.6 内核支持抢占调度，一个进程在内核执行的时候可能被另一高优先级进程打断，进程与抢占它的进程访问共享资源的情况类似于 SMP 的多个 CPU。

3. 中断（硬中断、软中断、Tasklet、底半部）与进程之间

中断可以打断正在执行的进程，如果中断处理程序访问进程正在访问的资源，则竞态也会发生。

此外，中断也有可能被新的更高优先级的中断打断，因此，多个中断之间本身也可能引起并发而导致竞态。

上述并发的发生情况除了 SMP 是真正的并行以外，其他的都是“宏观并行，微观串行”的，但其引发的实质问题和 SMP 相似。

解决竞态问题的途径是保证对共享资源的互斥访问，所谓互斥访问是指一个执行单元在访问共享资源的时候，其他的执行单元被禁止访问。

访问共享资源的代码区域称为临界区（critical sections），临界区需要被以某种互斥机制加以保护。中断屏蔽、原子操作、自旋锁和信号量等是 Linux 设备驱动中可采用的互斥途径，7.2~7.5 节将一一讲解这些方法。

7.2 中断屏蔽

在单 CPU 范围内避免竞态的一种简单而省事的方法是在进入临界区之前屏蔽系统的中断。CPU 一般都具备屏蔽中断和打开中断的功能，这项功能可以保证正在执行的内核执行路径不被中断处理程序所抢占，防止某些竞态条件的发生。具体而言，中断屏蔽将使得中断与进程之间的并发不再发生，而且，由于 Linux 内核的进程调度等操作都依赖中断来实现，内核抢占进程之间的并发也得以避免了。

中断屏蔽的使用方法为：

```
local_irq_disable() /* 屏蔽中断 */
...
critical section /* 临界区*/
...
local_irq_enable() /* 开中断*/
```

由于 Linux 的异步 I/O、进程调度等很多重要操作都依赖于中断，中断对于内核的运行非常重要，在屏蔽中断期间所有的中断都无法得到处理，因此长时间屏蔽中断是很危险的，有可能造成数据丢失乃至系统崩溃等后果。这就要求在屏蔽了中断之后，当前的内核执行路径应当尽快地执行完临界区的代码。

`local_irq_disable()`和 `local_irq_enable()`都只能禁止和使能本 CPU 内的中断，因此，并不能解决 SMP 多 CPU 引发的竞态。因此，单独使用中断屏蔽通常不是一种值得推荐的避免竞态的方法，它适宜与下文将要介绍的自旋锁联合使用。

与 `local_irq_disable()`不同的是，`local_irq_save(flags)`除了进行禁止中断的操作以外，还保存目前的 CPU 的中断位信息，`local_irq_restore(flags)`进行的是与 `local_irq_save(flags)`相反的操作。

如果只是禁止中断的底半部，应使用 `local_bh_disable()`，使能被 `local_bh_disable()`禁止的底半部应该调用 `local_bh_enable()`。



7.3 原子操作

原子操作指的是在执行过程中不会被别的代码路径所中断的操作。

Linux 内核提供了一系列函数来实现内核中的原子操作，这些函数又分为两类，分别针对位和整型变量进行原子操作。它们的共同点是在任何情况下操作都是原子的，内核代码可以安全地调用它们而不会被打断。位和整型变量原子操作都依赖底层 CPU 的原子操作来实现，因此所有这些函数都与 CPU 架构密切相关。

7.3.1 整型原子操作

1. 设置原子变量的值

```
void atomic_set(atomic_t *v, int i); /* 设置原子变量的值为 i */
atomic_t v = ATOMIC_INIT(0); /* 定义原子变量 v 并初始化为 0 */
```

2. 获取原子变量的值

```
atomic_read(atomic_t *v); /* 返回原子变量的值 */
```

3. 原子变量加/减

```
void atomic_add(int i, atomic_t *v); /* 原子变量增加 i */
void atomic_sub(int i, atomic_t *v); /* 原子变量减少 i */
```

4. 原子变量自增/自减

```
void atomic_inc(atomic_t *v); /* 原子变量增加 1 */
void atomic_dec(atomic_t *v); /* 原子变量减少 1 */
```

5. 操作并测试

```
int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
int atomic_sub_and_test(int i, atomic_t *v);
```

上述操作对原子变量执行自增、自减和减操作后（注意没有加）测试其是否为 0，为 0 返回 true，否则返回 false。

6. 操作并返回

```
int atomic_add_return(int i, atomic_t *v);
int atomic_sub_return(int i, atomic_t *v);
int atomic_inc_return(atomic_t *v);
int atomic_dec_return(atomic_t *v);
```

上述操作对原子变量进行加/减和自增/自减操作，并返回新的值。

7.3.2 位原子操作

1. 设置位

```
void set_bit(nr, void *addr);
```

上述操作设置 addr 地址的第 nr 位，所谓设置位即是将位写为 1。

2. 清除位

```
void clear_bit(nr, void *addr);
```

上述操作清除 addr 地址的第 nr 位，所谓清除位即是将位写为 0。

3. 改变位

```
void change_bit(nr, void *addr);
```

上述操作对 `addr` 地址的第 `nr` 位进行反置。

4. 测试位

```
test_bit(nr, void *addr);
```

上述操作返回 `addr` 地址的第 `nr` 位。

5. 测试并操作位

```
int test_and_set_bit(nr, void *addr);
int test_and_clear_bit(nr, void *addr);
int test_and_change_bit(nr, void *addr);
```

上述 `test_and_xxx_bit(nr, void *addr)` 操作等同于执行 `test_bit(nr, void *addr)` 后再执行 `xxx_bit(nr, void *addr)`。

代码清单 7.1 给出了原子变量的使用例子，它用于使得设备最多只能被一个进程打开。

代码清单 7.1 使用原子变量实现设备只能被一个进程打开

```
1 static atomic_t xxx_available = ATOMIC_INIT(1); /*定义原子变量*/
2
3 static int xxx_open(struct inode *inode, struct file *filp)
4 {
5     ...
6     if (!atomic_dec_and_test(&xxx_available)) {
7         atomic_inc(&xxx_available);
8         return - EBUSY; /*已经打开*/
9     }
10    ...
11    return 0; /* 成功 */
12 }
13
14 static int xxx_release(struct inode *inode, struct file *filp)
15 {
16     atomic_inc(&xxx_available); /* 释放设备 */
17     return 0;
18 }
```

7.4 自旋锁

7.4.1 自旋锁的使用

自旋锁（spin lock）是一种典型的对临界资源进行互斥访问的手段，其名称来源于它的工作方式。为了获得一个自旋锁，在某 CPU 上运行的代码需先执行一个原子操作，该操作测试并设置（test-and-set）某个内存变量，由于它是原子操作，所以在该操作完成之前其他执行单元不可能访问这个内存变量。如果测试结果表明锁已经空闲，则程序获得这个自旋锁并继续执行；如果测试结果表明锁仍被占用，程序将在一个小的循环内重复这个“测试并设置”操作，即进行所谓的“自旋”，通俗地说就是“在原地打转”。当自旋锁的持有者通过重置该变量释放这个自旋锁后，某个



等待的“测试并设置”操作向其调用者报告锁已释放。

理解自旋锁最简单的方法是把它作为一个变量看待，该变量把一个临界区或者标记为“我当前在运行，请稍等一会”或者标记为“我当前不在运行，可以被使用”。如果 A 执行单元首先进入例程，它将持有自旋锁；当 B 执行单元试图进入同一个例程时，将获知自旋锁已被持有，需等到 A 执行单元释放后才能进入。

Linux 中与自旋锁相关的操作主要有以下 4 种。

1. 定义自旋锁

```
spinlock_t lock;
```

2. 初始化自旋锁

```
spin_lock_init(&lock)
```

该宏用于动态初始化自旋锁 lock。

3. 获得自旋锁

```
spin_lock(&lock)
```

该宏用于获得自旋锁 lock，如果能够立即获得锁，它就马上返回，否则，它将自旋在那里，直到该自旋锁的保持者释放。

```
spin_trylock(&lock)
```

该宏尝试获得自旋锁 lock，如果能立即获得锁，它获得锁并返回真，否则立即返回假，实际上不再“在原地打转”。

4. 释放自旋锁

```
spin_unlock(&lock)
```

该宏释放自旋锁 lock，它与 spin_trylock 或 spin_lock 配对使用。

自旋锁一般这样被使用：

```
/* 定义一个自旋锁*/
spinlock_t lock;
spin_lock_init(&lock);

spin_lock (&lock) ; /* 获取自旋锁，保护临界区 */
. . . /* 临界区*/
spin_unlock (&lock) ; /* 解锁*/
```

自旋锁主要针对 SMP 或单 CPU 但内核可抢占的情况，对于单 CPU 和内核不支持抢占的系统，自旋锁退化为空操作。在单 CPU 和内核可抢占的系统中，自旋锁持有期间内核的抢占将被禁止。由于内核可抢占的单 CPU 系统的行为实际很类似于 SMP 系统，因此，在这样的单 CPU 系统中使用自旋锁仍十分必要。

尽管用了自旋锁可以保证临界区不受别的 CPU 和本 CPU 内的抢占进程打扰，但是得到锁的代码路径在执行临界区的时候，还可能受到中断和底半部（BH，稍后的章节会介绍）的影响。为了防止这种影响，就需要用到自旋锁的衍生。spin_lock()/spin_unlock()是自旋锁机制的基础，它们和关中断 local_irq_disable()/开中断 local_irq_enable()、关底半部 local_bh_disable()/开底半部 local_bh_enable()、关中断并保存状态字 local_irq_save()/开中断并恢复状态 local_irq_restore()结合就形成了整套自旋锁机制，关系如下：

```
spin_lock_irq() = spin_lock() + local_irq_disable()
spin_unlock_irq() = spin_unlock() + local_irq_enable()
spin_lock_irqsave() = spin_lock() + local_irq_save()
spin_unlock_irqrestore() = spin_unlock() + local_irq_restore()
```

```
spin_lock_bh() = spin_lock() + local_bh_disable()
spin_unlock_bh() = spin_unlock() + local_bh_enable()
```

`spin_lock_irq()`、`spin_lock_irqsave()`、`spin_lock_bh()`类似函数会为自旋锁的使用系好“安全带”以避免突入其来的中断驶入对系统造成的伤害。

驱动工程师应谨慎使用自旋锁，而且在使用中还要特别注意如下几个问题。

(1) 自旋锁实际上是忙等锁，当锁不可用时，CPU 一直循环执行“测试并设置”该锁直到可用而取得该锁，CPU 在等待自旋锁时不做任何有用的工作，仅仅是等待。因此，只有在占用锁的时间极短的情况下，使用自旋锁才是合理的。当临界区很大，或有共享设备的时候，需要较长时间占用锁，使用自旋锁会降低系统的性能。

(2) 自旋锁可能导致系统死锁。引发这个问题最常见的情况是递归使用一个自旋锁，即如果一个已经拥有某个自旋锁的 CPU 想第二次获得这个自旋锁，则该 CPU 将死锁。

(3) 自旋锁锁定期间不能调用可能引起进程调度的函数。如果进程获得自旋锁之后再阻塞，如调用 `copy_from_user()`、`copy_to_user()`、`kmalloc()`和 `msleep()`等函数，则可能导致内核的崩溃。

代码清单 7.2 给出了自旋锁的使用例子，它被用于实现使得设备只能被最多 1 个进程打开，等同于代码清单 7.1。

代码清单 7.2 使用自旋锁实现设备只能被一个进程打开

```
1 int xxx_count = 0; /*定义文件打开次数计数*/
2
3 static int xxx_open(struct inode *inode, struct file *filp)
4 {
5     ...
6     spinlock(&xxx_lock);
7     if (xxx_count) { /*已经打开*/
8         spin_unlock(&xxx_lock);
9         return -EBUSY;
10    }
11    xxx_count++; /*增加使用计数*/
12    spin_unlock(&xxx_lock);
13    ...
14    return 0; /* 成功 */
15 }
16
17 static int xxx_release(struct inode *inode, struct file *filp)
18 {
19     ...
20     spinlock(&xxx_lock);
21     xxx_count--; /*减少使用计数*/
22     spin_unlock(&xxx_lock);
23
24     return 0;
25 }
```

7.4.2 读写自旋锁

自旋锁不关心锁定的临界区究竟进行怎样的操作，不管是读还是写，它都一视同仁。即便多个执行单元同时读取临界资源也会被锁住。实际上，对共享资源并发访问时，多个执行单元同时读取它是不会有问题的，自旋锁的衍生锁读写自旋锁（`rwlock`）可允许读的并发。读写自旋锁是



一种比自旋锁粒度更小的锁机制，它保留了“自旋”的概念，但是在写操作方面，只能最多有 1 个写进程，在读操作方面，同时可以有多个读执行单元。当然，读和写也不能同时进行。

读写自旋锁涉及的操作如下。

1. 定义和初始化读写自旋锁

```
rwlock_t my_rwlock = RW_LOCK_UNLOCKED; /* 静态初始化 */
rwlock_t my_rwlock;
rwlock_init(&my_rwlock); /* 动态初始化 */
```

2. 读锁定

```
void read_lock(rwlock_t *lock);
void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
void read_lock_irq(rwlock_t *lock);
void read_lock_bh(rwlock_t *lock);
```

3. 读解锁

```
void read_unlock(rwlock_t *lock);
void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void read_unlock_irq(rwlock_t *lock);
void read_unlock_bh(rwlock_t *lock);
```

在对共享资源进行读取之前，应该先调用读锁定函数，完成之后应调用读解锁函数。

`read_lock_irqsave()`、`read_lock_irq()`和 `read_lock_bh()`也分别是 `read_lock()`分别与 `local_irq_save()`、`local_irq_disable()`和 `local_bh_disable()`的组合，读解锁函数 `read_unlock_irqrestore()`、`read_unlock_irq()`、`read_unlock_bh()`的情况与此类似。

4. 写锁定

```
void write_lock(rwlock_t *lock);
void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
void write_lock_irq(rwlock_t *lock);
void write_lock_bh(rwlock_t *lock);
int write_trylock(rwlock_t *lock);
```

5. 写解锁

```
void write_unlock(rwlock_t *lock);
void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void write_unlock_irq(rwlock_t *lock);
void write_unlock_bh(rwlock_t *lock);
```

`write_lock_irqsave()`、`write_lock_irq()`、`write_lock_bh()`分别是 `write_lock()`与 `local_irq_save()`、`local_irq_disable()`和 `local_bh_disable()`的组合，写解锁函数 `write_unlock_irqrestore()`、`write_unlock_irq()`、`write_unlock_bh()`的情况与此类似。

在对共享资源进行写之前，应该先调用写锁定函数，完成之后应调用写解锁函数。和 `spin_trylock()`一样，`write_trylock()`也只是尝试获取读写自旋锁，不管成功失败，都会立即返回。

读写自旋锁一般这样被使用：

```
rwlock_t lock; /* 定义 rwlock */
rwlock_init(&lock); /* 初始化 rwlock */

/* 读时获取锁 */
read_lock(&lock);
... /* 临界资源 */
read_unlock(&lock);

/* 写时获取锁 */
```

```
write_lock_irqsave(&lock, flags);
... /* 临界资源 */
write_unlock_irqrestore(&lock, flags);
```

7.4.3 顺序锁

顺序锁（seqlock）是对读写锁的一种优化，若使用顺序锁，读执行单元绝不会被写执行单元阻塞，也就是说，读执行单元在写执行单元对被顺序锁保护的共享资源进行写操作时仍然可以继续读，而不必等待写执行单元完成写操作，写执行单元也不需要等待所有读执行单元完成读操作才去进行写操作。

但是，写执行单元与写执行单元之间仍然是互斥的，即如果有写执行单元在进行写操作，其他写执行单元必须自旋在那里，直到写执行单元释放了顺序锁。如果读执行单元在读操作期间，写执行单元已经发生了写操作，那么，读执行单元必须重新读取数据，以便确保得到的数据是完整的。这种锁对于读写同时进行的概率比较小的情况，性能是非常好的，而且它允许读写同时进行，因而更大地提高了并发性。

顺序锁有一个限制，它必须要求被保护的共享资源不含有指针，因为写执行单元可能使得指针失效，但读执行单元如果正要访问该指针，将导致 oops。

在 Linux 内核中，写执行单元涉及的顺序锁操作如下。

1. 获得顺序锁

```
void write_seqlock(seqlock_t *sl);
int write_tryseqlock(seqlock_t *sl);
write_seqlock_irqsave(lock, flags)
write_seqlock_irq(lock)
write_seqlock_bh(lock)
```

其中：

```
write_seqlock_irqsave() = local_irq_save() + write_seqlock()
write_seqlock_irq() = local_irq_disable() + write_seqlock()
write_seqlock_bh() = local_bh_disable() + write_seqlock()
```

2. 释放顺序锁

```
void write_sequnlock(seqlock_t *sl);
write_sequnlock_irqrestore(lock, flags)
write_sequnlock_irq(lock)
write_sequnlock_bh(lock)
```

其中：

```
write_sequnlock_irqrestore() = write_sequnlock() + local_irq_restore()
write_sequnlock_irq() = write_sequnlock() + local_irq_enable()
write_sequnlock_bh() = write_sequnlock() + local_bh_enable()
```

写执行单元使用顺序锁的模式如下：

```
write_seqlock(&seqlock_a);
.../* 写操作代码块 */
write_sequnlock(&seqlock_a);
```

因此，对写执行单元而言，它的使用与 spinlock 相同。

读执行单元涉及的顺序锁操作如下。

1. 读开始

```
unsigned read_seqbegin(const seqlock_t *sl);
read_seqbegin_irqsave(lock, flags)
```



读执行单元在对被顺序锁 `s1` 保护的共享资源进行访问前需要调用该函数, 该函数仅返回顺序锁 `s1` 的当前顺序号。其中:

```
read_seqbegin_irqsave() = local_irq_save() + read_seqbegin()
```

2. 重读

```
int read_seqretry(const seqlock_t *sl, unsigned iv);
read_seqretry_irqrestore(lock, iv, flags)
```

读执行单元在访问完被顺序锁 `s1` 保护的共享资源后需要调用该函数来检查, 在读访问期间是否有写操作。如果有写操作, 读执行单元就需要重新进行读操作。其中:

```
read_seqretry_irqrestore() = read_seqretry() + local_irq_restore()
```

读执行单元使用顺序锁的模式如下:

```
do {
    seqnum = read_seqbegin(&seqlock_a);
    /* 读操作代码块 */
    ...
} while (read_seqretry(&seqlock_a, seqnum));
```

7.4.4 读-拷贝-更新

RCU (Read-Copy Update, 读-拷贝-更新), 它是基于其原理命名的。RCU 并不是新的锁机制, 它只是对 Linux 内核而言是新的。早在 20 世纪 80 年代就有了这种机制, 而在 Linux 中是在开发内核 2.5.43 中引入该技术的并正式包含在 2.6 内核中。

对于被 RCU 保护的共享数据结构, 读执行单元不需要获得任何锁就可以访问它, 不使用原子指令, 而且在除 alpha 的所有架构上也不需要内存屏障 (Memory Barrier), 因此不会导致锁竞争、内存延迟以及流水线停滞。不需要锁也使得使用更容易, 因为死锁问题就不需要考虑了。使用 RCU 的写执行单元在访问它前需首先拷贝一个副本, 然后对副本进行修改, 最后使用一个回调机制在适当的时机把指向原来数据的指针重新指向新的被修改的数据, 这个时机就是所有引用该数据的 CPU 都退出对共享数据的操作的时候。读执行单元没有任何同步开销, 而写执行单元的同步开销则取决于使用的写执行单元间同步机制。

RCU 可以看作读写锁的高性能版本, 相比读写锁, RCU 的优点在于既允许多个读执行单元同时访问被保护的数据, 又允许多个读执行单元和多个写执行单元同时访问被保护的数据。但是, RCU 不能替代读写锁, 因为如果写比较多时, 对读执行单元的性能提高不能弥补写执行单元导致的损失。因为使用 RCU 时, 写执行单元之间的同步开销会比较大, 它需要延迟数据结构的释放, 复制被修改的数据结构, 它也必须使用某种锁机制同步并行的其他写执行单元的修改操作。

Linux 中提供的 RCU 操作包括如下 4 种。

1. 读锁定

```
rcu_read_lock()
rcu_read_lock_bh()
```

2. 读解锁

```
rcu_read_unlock()
rcu_read_unlock_bh()
```

使用 RCU 进行读的模式如下:

```
rcu_read_lock()
.../* 读临界区*/
rcu_read_unlock()
```


其中 `rcu_read_lock()` 和 `rcu_read_unlock()` 实质只是禁止和使能内核的抢占调度：

```
#define rcu_read_lock()      preempt_disable()
#define rcu_read_unlock()    preempt_enable()
```

其变种 `rcu_read_lock_bh()`、`rcu_read_unlock_bh()` 则定义为：

```
#define rcu_read_lock_bh()    local_bh_disable()
#define rcu_read_unlock_bh()  local_bh_enable()
```

3. 同步 RCU

```
synchronize_rcu()
```

该函数由 RCU 写执行单元调用，它将阻塞写执行单元，直到所有的读执行单元已经完成读执行单元临界区，写执行单元才可以继续下一步操作。如果有多个 RCU 写执行单元调用该函数，它们将在一个 `grace period`（即所有的读执行单元已经完成对临界区的访问）之后全部被唤醒。`synchronize_rcu()` 保证所有 CPU 都处理完正在运行的读执行单元临界区。

```
synchronize_kernel()
```

内核代码使用该函数来等待所有 CPU 处于可抢占状态，目前功能等同于 `synchronize_rcu()`，但现在已经不建议使用，而是使用 `synchronize_sched()`，该函数用于等待所有 CPU 都处在可抢占状态，它能保证正在运行的中断处理函数处理完毕，但不能保证正在运行的软中断处理完毕。

● 挂接回调

```
void call_rcu(struct rcu_head *head,
              void (*func)(struct rcu_head *rcu));
```

函数 `call_rcu()` 也由 RCU 写执行单元调用，它不会使写执行单元阻塞，因而可以在中断上下文或软中断使用。该函数将把函数 `func` 挂接到 RCU 回调函数链上，然后立即返回。函数 `synchronize_rcu()` 的实现实际上使用了 `call_rcu()` 函数。

```
void call_rcu_bh(struct rcu_head *head,
                 void (*func)(struct rcu_head *rcu));
```

`call_rcu_bh()` 函数的功能几乎与 `call_rcu()` 完全相同，惟一差别就是它把软中断的完成也当作经历一个 `quiescent state`（静默状态），因此如果写执行单元使用了该函数，在进程上下文的读执行单元必须使用 `rcu_read_lock_bh()`。

每个 CPU 维护两个数据结构 `rcu_data` 和 `rcu_bh_data`，它们用于保存回调函数，函数 `call_rcu()` 把回调函数注册到 `rcu_data`，而 `call_rcu_bh()` 则把回调函数注册到 `rcu_bh_data`，在每一个数据结构上，回调函数被组成一个链表，先注册的排在前头，后注册的排在末尾。

使用 RCU 时，读执行单元必须提供一个信号给写执行单元以便写执行单元能够确定数据可以被安全地释放或修改的时机。有一个专门的垃圾收集器来探测读执行单元的信号，一旦所有的读执行单元都已经发送信号告知它们都不在使用被 RCU 保护的数据结构，垃圾收集器就调用回调函数完成最后的数据释放或修改操作。

RCU 还增加了链表操作函数的 RCU 版本：

```
static inline void list_add_rcu(struct list_head *new, struct list_head *head);
```

该函数把链表元素 `new` 插入 RCU 保护的链表 `head` 的开头，内存栅保证了在引用这个新插入的链表元素之前，新链表元素的链接指针的修改对所有读执行单元是可见的。

```
static inline void list_add_tail_rcu(struct list_head *new,
                                     struct list_head *head);
```

该函数类似于 `list_add_rcu()`，它将把新的链表元素 `new` 添加到被 RCU 保护的链表的末尾。

```
static inline void list_del_rcu(struct list_head *entry);
```



该函数从 RCU 保护的链表中删除指定的链表元素 `entry`。

```
static inline void list_replace_rcu(struct list_head *old, struct list_head *new);
```

该函数是 RCU 新添加的函数，并不存在非 RCU 版本。它使用新的链表元素 `new` 取代旧的链表元素 `old`，内存栅保证在引用新的链表元素之前，它对链接指针的修正对所有读执行单元是可见的。

```
list_for_each_rcu(pos, head)
```

该宏用于遍历由 RCU 保护的链表 `head`，只要在读执行单元临界区使用该函数，它就可以安全地和其他 `_rcu` 链表操作函数，如 `list_add_rcu()` 并发运行。

```
list_for_each_safe_rcu(pos, n, head)
```

该宏类似于 `list_for_each_rcu`，但不同之处在于它允许安全地删除当前链表元素 `pos`。

```
list_for_each_entry_rcu(pos, head, member)
```

该宏类似于 `list_for_each_rcu`，不同之处在于它用于遍历指定类型的数据结构链表，当前链表元素 `pos` 为一包含 `struct list_head` 结构的特定的数据结构。

```
static inline void hlist_del_rcu(struct hlist_node *n)
```

它从由 RCU 保护的哈希链表中移走链表元素 `n`。

```
static inline void hlist_add_head_rcu(struct hlist_node *n,  
    struct hlist_head *h);
```

该函数用于把链表元素 `n` 插入被 RCU 保护的哈希链表的开头，但同时允许读执行单元对该哈希链表的遍历。内存栅确保在引用新链表元素之前，它对指针的修改对所有读执行单元可见。

```
hlist_for_each_rcu(pos, head)
```

该宏用于遍历由 RCU 保护的哈希链表 `head`，只要在读端临界区使用该函数，它就可以安全地和其他 `_rcu` 哈希链表操作函数（如 `hlist_add_rcu`）并发运行。

```
hlist_for_each_entry_rcu(tpos, pos, head, member)
```

类似于 `hlist_for_each_rcu()`，不同之处在于它用于遍历指定类型的数据结构哈希链表，当前链表元素 `pos` 为一包含 `struct list_head` 结构的特定的数据结构。

目前，RCU 的使用在内核中已经非常普遍，内核中大量原先使用读写锁的代码被 RCU 替换，下面的表单左右两列平行地分别给出使用读写锁和 RCU 实现链表元素读、删除、添加和修改的代码：

<pre>/* 原先的 audit_filter_task 函数，读链表前获得 读写锁 */ static enum audit_state audit_filter_task(struct task_struct *tsk) { struct audit_entry *e; enum audit_state state; read_lock(&auditsc_lock); /* 遍历链表 */ list_for_each_entry(e, &audit_tsklist, list) { if (audit_filter_rules(tsk, &e ->rule, NULL, &state)) { read_unlock(&auditsc_lock); return state; } } }</pre>	<pre>/* 修改后的 audit_filter_task 函数，采用 RCU */ static enum audit_state audit_filter_task(struct task_struct *tsk) { struct audit_entry *e; enum audit_state state; rcu_read_lock(); /* 遍历链表 */ list_for_each_entry_rcu(e, &audit_tsklist, list) { if (audit_filter_rules(tsk, &e ->rule, NULL, &state)) { rcu_read_unlock(); return state; } } }</pre>
---	--

<pre> } } read_unlock(&auditsc_lock); return AUDIT_BUILD_CONTEXT; } /* 原先的 audit_del_rule, 删除链表元素前获得读写锁 */ static inline int audit_del_rule(struct audit_rule *rule, struct list_head*list) { struct audit_entry *e; write_lock(&auditsc_lock); /* 遍历链表 */ list_for_each_entry(e, list, list) { if (!audit_compare_rule(rule, &e ->rule)) { list_del(&e->list); /* 删除链表元素 */ write_unlock(&auditsc_lock); return 0; } } write_unlock(&auditsc_lock); return - EFAULT; } /* 原先的 audit_add_rule, 添加链表元素前获得读写锁 */ static inline int audit_add_rule(struct audit_entry *entry, struct list_head*list) { write_lock(&auditsc_lock); if (entry->rule.flags &AUDIT_PREPEND) { entry->rule.flags &= ~AUDIT_PREPEND; list_add(&entry->list, list); } else { list_add_tail(&entry->list, list); } write_unlock(&auditsc_lock); return 0; } /* 原先的 audit_upd_rule 函数, 在修改链表元素前获得读写锁 */ static inline int audit_upd_rule(struct audit_rule *rule, struct list_head *list, __u32 newaction, __u32 newfield_count) { struct audit_entry *e; struct audit_newentry *ne; write_lock(&auditsc_lock); /* 遍历链表 */ </pre>	<pre> } } rcu_read_unlock(); return AUDIT_BUILD_CONTEXT; } /* 修改后的 audit_del_rule, 使用 list_del_rcu 删除链表元素, 并调用 call_rcu 注册回调函数 */ static inline int audit_del_rule(struct audit_rule *rule, struct list_head*list) { struct audit_entry *e; /* 遍历链表 */ list_for_each_entry(e, list, list) { if (!audit_compare_rule(rule, &e ->rule)) { list_del_rcu(&e->list); call_rcu(&e->rcu, audit_free_rule); return 0; } } return - EFAULT; } /* 修改后的 audit_add_rule, 在添加链表元素时 使用 list_add_xxx 函数 */ static inline int audit_add_rule(struct audit_entry *entry, struct list_head*list) { if (entry->rule.flags &AUDIT_PREPEND) { entry->rule.flags &= ~AUDIT_PREPEND; list_add_rcu(&entry->list, list); } else { list_add_tail_rcu(&entry->list, list); } return 0; } /* 修改后的 audit_upd_rule, 使用 list_replace_ rcu 修改链表元素, 并用 call_rcu 注册回调函数 */ static inline int audit_upd_rule(struct audit_rule *rule, struct list_head *list, __u32 newaction, __u32 newfield_count) { struct audit_entry *e; struct audit_newentry *ne; </pre>
---	--



```
list_for_each_entry(e, list, list) {
    if (!audit_compare_rule(rule, &e
        ->rule)) {
        e->rule.action = newaction;
        e->rule.file_count =
            newfield_count;
        write_unlock(&auditsc_lock);

        return 0;
    }
}
write_unlock(&auditsc_lock);
return - EFAULT;
}
```

```
/* 遍历链表 */
list_for_each_entry(e, list, list) {
    if (!audit_compare_rule(rule, &e
        ->rule)) {
        ne = kmalloc(sizeof(*entry),
            GFP_ATOMIC); /* 分配新元素内存 */
        if (ne == NULL)
            return - ENOMEM;
        audit_copy_rule(&ne->rule, &e
            ->rule); /* 写前拷贝 */
        ne->rule.action = newaction;
        ne->rule.file_count =
            newfield_count;
        list_replace_rcu(e, ne);
        call_rcu(&e->rcu, audit_free_rule);
        return 0;
    }
}

return - EFAULT;
}
```

7.5 信号量

7.5.1 信号量的使用

信号量 (semaphore) 是用于保护临界区的一种常用方法, 它的使用方式和自旋锁类似。与自旋锁相同, 只有得到信号量的进程才能执行临界区代码。但是, 与自旋锁不同的是, 当获取不到信号量时, 进程不会原地打转而是进入休眠等待状态。

Linux 中与信号量相关的操作主要有:

1. 定义信号量

下列代码定义名称为 `sem` 的信号量:

```
struct semaphore sem;
```

2. 初始化信号量

```
void sema_init(struct semaphore *sem, int val);
```

该函数初始化信号量, 并设置信号量 `sem` 的值为 `val`。尽管信号量可以被初始化为大于 1 的值从而成为一个计数信号量, 但是它通常不被这样使用。

```
#define init_MUTEX(sem) sema_init(sem, 1)
```

该宏用于初始化一个用于互斥的信号量, 它把信号量 `sem` 的值设置为 1;

```
#define init_MUTEX_LOCKED(sem) sema_init(sem, 0)
```

该宏也用于初始化一个信号量, 但它把信号量 `sem` 的值设置为 0;

此外, 下面两个宏是定义并初始化信号量的“快捷方式”:

```
DECLARE_MUTEX(name)
```

```
DECLARE_MUTEX_LOCKED(name)
```

前者定义一个名为 `name` 的信号量并初始化为 1；后者定义一个名为 `name` 的信号量并初始化为 0。

3. 获得信号量

```
void down(struct semaphore * sem);
```

该函数用于获得信号量 `sem`，它会导致睡眠，因此不能在中断上下文使用；

```
int down_interruptible(struct semaphore * sem);
```

该函数功能与 `down` 类似，不同之处为，因为 `down()` 而进入睡眠状态的进程不能被信号打断，但因为 `down_interruptible()` 而进入睡眠状态的进程能被信号打断，信号也会导致该函数返回，这时候函数的返回值非 0；

```
int down_trylock(struct semaphore * sem);
```

该函数尝试获得信号量 `sem`，如果能够立刻获得，它就获得该信号量并返回 0，否则，返回非 0 值。它不会导致调用者睡眠，可以在中断上下文使用。

在使用 `down_interruptible()` 获取信号量时，对返回值一般会进行检查，如果非 0，通常立即返回 `- ERESTARTSYS`，如：

```
if (down_interruptible(&sem))
    return - ERESTARTSYS;
```

4. 释放信号量

```
void up(struct semaphore * sem);
```

该函数释放信号量 `sem`，唤醒等待者。

信号量一般这样被使用：

```
/* 定义信号量
DECLARE_MUTEX(mount_sem);
down(&mount_sem); /* 获取信号量，保护临界区
...
critical section /* 临界区
...
up(&mount_sem); /* 释放信号量
```



Linux 自旋锁和信号量所采用的“获取锁—访问临界区—释放锁”的方式，姑且称之为“互斥三部曲”，实际存在于几乎所有的多任务操作系统之中，在 WIN32、VxWorks 等中皆如此。

代码清单 7.3 给出了使用信号量实现设备只能被一个进程打开的例子，等同于代码清单 7.1 和 7.2。

代码清单 7.3 使用信号量实现设备只能被一个进程打开

```
1 static DECLARE_MUTEX(xxx_lock); /* 定义互斥锁
2
3 static int xxx_open(struct inode *inode, struct file *filp)
4 {
5     ...
6     if (down_trylock(&xxx_lock)) /* 获得打开锁
7         return - EBUSY; /* 设备忙
8     ...
9     return 0; /* 成功 */
```



```
10 }
11
12 static int xxx_release(struct inode *inode, struct file *filp)
13 {
14     up(&xxx_lock); /* 释放打开锁 */
15     return 0;
16 }
```

7.5.2 信号量用于同步

如果信号量被初始化为 0，则它可以用于同步，同步意味着一个执行单元的继续执行需等待另一执行单元完成某事，保证执行的先后顺序。如图 7.4 所示，执行单元 A 执行代码区域 b 之前，必须等待执行单元 B 执行完代码单元 c，信号量可辅助这一同步过程。

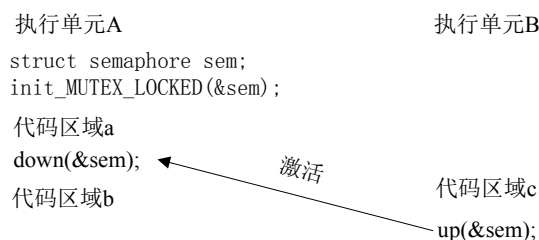


图 7.4 信号量用于同步

7.5.3 完成量用于同步

Linux 提供了一种比 7.5.2 所述更好的同步机制，即完成量（completion，关于这个名词，至今没有好的翻译，笔者将其译为“完成量”），它用于一个执行单元等待另一个执行单元执行完某事。

Linux 中与 completion 相关的操作主要有以下 4 种。

1. 定义完成量

下列代码定义名为 my_completion 的完成量：

```
struct completion my_completion;
```

2. 初始化 completion

下列代码初始化 my_completion 这个完成量：

```
init_completion(&my_completion);
```

对 my_completion 的定义和初始化可以通过如下快捷方式实现：

```
DECLARE_COMPLETION(my_completion);
```

3. 等待完成量

下列函数用于等待一个 completion 被唤醒：

```
void wait_for_completion(struct completion *c);
```

4. 唤醒完成量

下面两个函数用于唤醒完成量：

```
void complete(struct completion *c);
```

```
void complete_all(struct completion *c);
```

前者只唤醒一个等待的执行单元，后者释放所有等待同一完成量的执行单元。

图 7.5 描述了使用完成量实现的与图 7.4 对应的信号量实现的同步功能。

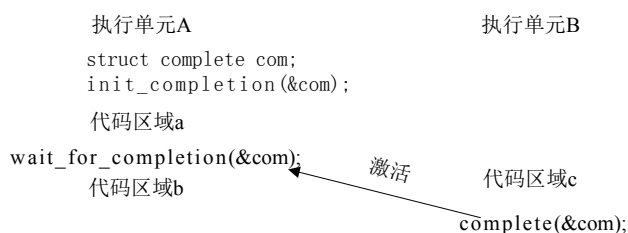


图 7.5 完成量用于同步

7.5.4 自旋锁 vs 信号量

自旋锁和信号量都是解决互斥问题的基本手段，面对特定的情况，应该如何取舍这两种手段呢？选择的依据是临界区的性质和系统的特点。

从严格意义上说，信号量和自旋锁属于不同层次的互斥手段，前者的实现有赖于后者。在信号量本身的实现上，为了保证信号量结构存取的原子性，在多 CPU 中需要自旋锁来互斥。

信号量是进程级的，用于多个进程之间对资源的互斥，虽然也是在内核中，但是该内核执行路径是以进程的身份，代表进程来争夺资源的。如果竞争失败，会发生进程上下文切换，当前进程进入睡眠状态，CPU 将运行其他进程。鉴于进程上下文切换的开销也很大，因此，只有当进程占用资源时间较长时，用信号量才是较好的选择。

当所要保护的临界区访问时间比较短时，用自旋锁是非常方便的，因为它节省上下文切换的时间。但是 CPU 得不到自旋锁会在那里空转直到其他执行单元解锁为止，所以要求锁不能在临界区里长时间停留，否则会降低系统的效率。

由此，可以总结出自旋锁和信号量选用的 3 项原则。

(1) 当锁不能被获取到时，使用信号量的开销是进程上下文切换时间 T_{sw} ，使用自旋锁的开销是等待获取自旋锁（由临界区执行时间决定） T_{cs} ，若 T_{cs} 比较小，宜使用自旋锁，若 T_{cs} 很大，应使用信号量。

(2) 信号量所保护的临界区可包含可能引起阻塞的代码，而自旋锁则绝对要避免用来保护包含这样代码的临界区。因为阻塞意味着要进行进程的切换，如果进程被切换出去后，另一个进程企图获取本自旋锁，死锁就会发生。

(3) 信号量存在于进程上下文，因此，如果被保护的共享资源需要在中断或软中断情况下使用，则在信号量和自旋锁之间只能选择自旋锁。当然，如果一定要使用信号量，则只能通过 `down_trylock()` 方式进行，不能获取就立即返回以避免阻塞。

7.5.5 读写信号量

读写信号量与信号量的关系与读写自旋锁和自旋锁的关系类似，读写信号量可能引起进程阻塞，但它可允许 N 个读执行单元同时访问共享资源，而最多只能有 1 个写执行单元。因此，读写信号量是一种相对放宽条件的粒度稍大于信号量的互斥机制。



读写自旋锁涉及的操作包括如下 5 种。

1. 定义和初始化读写信号量

```
struct rw_semaphore my_rws; /*定义读写信号量*/  
void init_rwsem(struct rw_semaphore *sem); /*初始化读写信号量*/
```

2. 读信号量获取

```
void down_read(struct rw_semaphore *sem);  
int down_read_trylock(struct rw_semaphore *sem);
```

3. 读信号量释放

```
void up_read(struct rw_semaphore *sem);
```

4. 写信号量获取

```
void down_write(struct rw_semaphore *sem);  
int down_write_trylock(struct rw_semaphore *sem);
```

5. 写信号量释放

```
void up_write(struct rw_semaphore *sem);
```

读写信号量一般这样被使用：

```
rw_semaphore rw_sem; /* 定义读写信号量 */  
init_rwsem(&rw_sem); /* 初始化读写信号量 */  
  
/* 读时获取信号量 */  
down_read(&rw_sem);  
... /* 临界资源*/  
up_read(&rw_sem);  
  
/* 写时获取信号量 */  
down_write(&rw_sem);  
... /* 临界资源 */  
up_write(&rw_sem);
```

7.6 互斥体

尽管信号量已经可以实现互斥的功能，而且包含 `DECLARE_MUTEX()`、`init_MUTEX ()` 等定义信号量的宏或函数，从名字上看就体现出了互斥体的概念，但是“正宗”的 `mutex` 在 Linux 内核中还是真实地存在着。

下面代码定义名为 `my_mutex` 的互斥体并初始化它：

```
struct mutex my_mutex;  
mutex_init(&my_mutex);
```

下面的两个函数用于获取互斥体：

```
void inline __sched mutex_lock(struct mutex *lock);  
int __sched mutex_lock_interruptible(struct mutex *lock);  
int __sched mutex_trylock(struct mutex *lock);
```

`mutex_lock()` 与 `mutex_lock_interruptible()` 的区别和 `down()` 与 `down_trylock()` 的区别完全一致，前者引起的睡眠不能被信号打断，而后者可以。`mutex_trylock()` 用于尝试获得 `mutex`，获取不到 `mutex` 时不会引起进程睡眠。

下列函数用于释放互斥体：

```
void __sched mutex_unlock(struct mutex *lock);
```


mutex 的使用方法和信号量用于互斥的场合完全一样:

```
struct mutex my_mutex; /* 定义 mutex */
mutex_init(&my_mutex); /* 初始化 mutex */

mutex_lock(&my_mutex); /* 获取 mutex */
.../* 临界资源*/
mutex_unlock(&my_mutex); /* 释放 mutex */
```

7.7 增加并发控制后的 globalmem 驱动

在 globalmem() 的读写函数中, 由于要调用 copy_from_user()、copy_to_user() 这些可能导致阻塞的函数, 因此不能使用自旋锁, 宜使用信号量。

驱动工程师习惯将某设备所使用的自旋锁、信号量等辅助手段也放在设备结构中, 因此, 可如代码清单 7.4 那样修改 globalmem_dev 结构体的定义, 并在模块初始化函数中初始化这个信号量, 如代码清单 7.5 所示。

代码清单 7.4 增加并发控制后的 globalmem 设备结构体

```
1 struct globalmem_dev {
2     struct cdev cdev; /*cdev 结构体*/
3     unsigned char mem[GLOBALMEM_SIZE]; /*全局内存*/
4     struct semaphore sem; /*并发控制用的信号量*/
5 };
```

代码清单 7.5 增加并发控制后的 globalmem 设备驱动模块加载函数

```
1 int globalmem_init(void)
2 {
3     int result;
4     dev_t devno = MKDEV(globalmem_major, 0);
5
6     /* 申请设备号*/
7     if (globalmem_major)
8         result = register_chrdev_region(devno, 1, "globalmem");
9     else { /* 动态申请设备号 */
10         result = alloc_chrdev_region(&devno, 0, 1, "globalmem");
11         globalmem_major = MAJOR(devno);
12     }
13     if (result < 0)
14         return result;
15
16     /* 动态申请设备结构体的内存*/
17     globalmem_devp = kmalloc(sizeof(struct globalmem_dev), GFP_KERNEL);
18     if (!globalmem_devp) { /*申请失败*/
19         result = - ENOMEM;
20         goto fail_malloc;
21     }
22     memset(globalmem_devp, 0, sizeof(struct globalmem_dev));
23
24     globalmem_setup_cdev(globalmem_devp, 0);
25     init_MUTEX(&globalmem_devp->sem); /*初始化信号量*/
```



```
26 return 0;
27
28 fail_malloc: unregister_chrdev_region(devno, 1);
29 return result;
30 }
```

在访问 `globalmem_dev` 中的共享资源时, 需先获取这个信号量, 访问完成后, 随即释放这个信号量。驱动中新的 `globalmem` 读、写操作如代码清单 7.6 所示。

代码清单 7.6 增加并发控制后的 `globalmem` 读写操作

```
1 /*增加并发控制后的 globalmem 读函数*/
2 static ssize_t globalmem_read(struct file *filp, char __user *buf, size_t size,
3     loff_t *ppos)
4 {
5     unsigned long p = *ppos;
6     unsigned int count = size;
7     int ret = 0;
8     struct globalmem_dev *dev = filp->private_data; /*获得设备结构体指针*/
9
10    /*分析和获取有效的写长度*/
11    if (p >= GLOBALMEM_SIZE)
12        return 0;
13    if (count > GLOBALMEM_SIZE - p)
14        count = GLOBALMEM_SIZE - p;
15
16    if (down_interruptible(&dev->sem)) /* 获得信号量*/
17        return - ERESTARTSYS;
18
19    /*内核空间→用户空间*/
20    if (copy_to_user(buf, (void*)(dev->mem + p), count)) {
21        ret = - EFAULT;
22    } else {
23        *ppos += count;
24        ret = count;
25
26        printk(KERN_INFO "read %d bytes(s) from %d\n", count, p);
27    }
28    up(&dev->sem); /* 释放信号量*/
29
30    return ret;
31 }
32
33 /*增加并发控制后的 globalmem 写函数*/
34 static ssize_t globalmem_write(struct file *filp, const char __user *buf,
35     size_t size, loff_t *ppos)
36 {
37     unsigned long p = *ppos;
38     unsigned int count = size;
39     int ret = 0;
40     struct globalmem_dev *dev = filp->private_data; /*获得设备结构体指针*/
41
42    /*分析和获取有效的写长度*/
43    if (p >= GLOBALMEM_SIZE)
44        return 0;
```

```

45  if (count > GLOBALMEM_SIZE - p)
46      count = GLOBALMEM_SIZE - p;
47
48  if (down_interruptible(&dev->sem)) /* 获得信号量 */
49      return - ERESTARTSYS;
50
51  /*用户空间→内核空间*/
52  if (copy_from_user(dev->mem + p, buf, count))
53      ret = - EFAULT;
54  else {
55      *ppos += count;
56      ret = count;
57
58      printk(KERN_INFO "written %d bytes(s) from %d\n", count, p);
59  }
60  up(&dev->sem); /* 释放信号量*/
61  return ret;
62 }

```

代码第 16~17 行和第 49~50 行用于获取信号量，如果 `down_interruptible()` 返回值非 0，则意味着其在获得信号量之前已被打断，这时写函数返回-`ERESTARTSYS`。代码第 28 和 60 行用于在对临界资源访问结束后释放信号量。

除了 `globalmem` 的读写操作之外，如果在读写的时候，另一执行单元执行 `MEM_CLEAR` IO 控制命令，也会导致全局内存的混乱，因此，`globalmem_ioctl()` 函数也需被重写，如代码清单 7.7 所示。

代码清单 7.7 增加并发控制后的 `globalmem` 设备驱动 `ioctl()` 函数

```

1  static int globalmem_ioctl(struct inode *inodep, struct file *filp, unsigned
2  int cmd, unsigned long arg)
3  {
4      struct globalmem_dev *dev = filp->private_data; /*获得设备结构体指针*/
5
6      switch (cmd) {
7          case MEM_CLEAR:
8              if (down_interruptible(&dev->sem)) /*获得信号量*/
9                  return - ERESTARTSYS;
10
11              memset(dev->mem, 0, GLOBALMEM_SIZE);
12              up(&dev->sem); /*释放信号量*/
13
14              printk(KERN_INFO "globalmem is set to zero\n");
15              break;
16
17          default:
18              return - EINVAL;
19      }
20      return 0;
21 }

```

增加并发控制后 `globalmem` 的完整驱动位于虚拟机的 `/home/lihacker/develop/svn/ldd6410-read-only/training/kernel/drivers/globalmem/ch7` 目录，其使用方法与 6.4 节 `globalmem` 驱动在用户空间的验证一致。



7.8 总结

并发和竞态广泛存在，中断屏蔽、原子操作、自旋锁和信号量都是解决并发问题的机制。中断屏蔽很少单独被使用，原子操作只能针对整数进行，因此自旋锁和信号量应用最为广泛。

自旋锁会导致死循环，锁定期间不允许阻塞，因此要求锁定的临界区小。信号量允许临界区阻塞，可以适用于临界区大的情况。

读写自旋锁和读写信号量分别是放宽了条件的自旋锁和信号量，它们允许多个执行单元对共享资源的并发读。

LINUX

第8章

Linux 设备驱动中的阻塞与非阻塞 I/O

本章导读

阻塞和非阻塞 I/O 是设备访问的两种不同模式，驱动程序可以灵活地支持用户空间对设备的这两种访问方式。

8.1 节讲述了阻塞和非阻塞 I/O 的区别，并讲解了实现阻塞 I/O 的等待队列机制，以及在 `globalfifo` 设备驱动中增加对阻塞 I/O 支持的方法，并进行了用户空间的验证。

8.2 节讲述了设备驱动的轮询（poll）操作的概念和编程方法，poll 可以帮助用户了解是否能对设备进行无阻塞的访问。

8.3 节讲解在 `globalfifo` 中增加 poll 操作的方法，并进行了用户空间的验证。





8.1 阻塞与非阻塞 I/O

阻塞操作是指在执行设备操作时，若不能获得资源，则挂起进程直到满足可操作的条件后再进行操作。被挂起的进程进入休眠状态，被从调度器的运行队列移走，直到等待的条件被满足。而非阻塞操作的进程在不能进行设备操作时，并不挂起，它或者放弃，或者不停地查询，直至可以进行操作为止。

驱动程序通常需要提供这样的能力：当应用程序进行 `read()`、`write()` 等系统调用时，若设备的资源不能获取，而用户又希望以阻塞的方式访问设备，驱动程序应在设备驱动的 `xxx_read()`、`xxx_write()` 等操作中将进程阻塞直到资源可以获取，此后，应用程序的 `read()`、`write()` 等调用才返回，整个过程仍然进行了正确的设备访问，用户并没有感知到；若用户以非阻塞的方式访问设备文件，则当设备资源不可获取时，设备驱动的 `xxx_read()`、`xxx_write()` 等操作应立即返回，`read()`、`write()` 等系统调用也随即被返回。

阻塞从字面上听起来似乎意味着低效率，实则不然，如果设备驱动不阻塞，则用户想获取设备资源只能不停地查询，这反而会无谓地耗费 CPU 资源。而阻塞访问时，不能获取资源的进程将进入休眠，它将 CPU 资源“礼让”给其他进程。

因为阻塞的进程会进入休眠状态，因此，必须确保有一个地方能够唤醒休眠的进程，否则，进程就真的“寿终正寝”了。唤醒进程的地方最大可能发生在中断里面，因为硬件资源获得的同时往往伴随着一个中断。

代码清单 8.1 和 8.2 分别演示了以阻塞和非阻塞方式读取串口一个字符的代码。实际的串口编程中，若使用非阻塞模式，还可借助信号（`sigaction`）以异步方式访问串口以提高 CPU 利用率，而这里仅仅是为了说明阻塞与非阻塞的区别。

代码清单 8.1 阻塞地读串口一个字符

```
char buf;
fd = open("/dev/ttyS1", O_RDWR);
...
res = read(fd, &buf, 1); /* 当串口上有输入时才返回 */
if(res==1)
printf("%c\n", buf);
```

代码清单 8.2 非阻塞地读串口一个字符

```
char buf;
fd = open("/dev/ttyS1", O_RDWR | O_NONBLOCK);
...
while(read(fd, &buf, 1) != 1)
continue; /* 串口上无输入也返回，所以要循环尝试读取串口 */
printf("%c\n", buf);
```

8.1.1 等待队列

在 Linux 驱动程序中，可以使用等待队列（`wait queue`）来实现阻塞进程的唤醒。`wait queue`

很早就作为一个基本的功能单位出现在 Linux 内核里了，它以队列为基础数据结构，与进程调度机制紧密结合，能够用于实现内核中的异步事件通知机制。等待队列可以用来同步对系统资源的访问，第 7 章中所讲述的信号量在内核中也依赖等待队列来实现。

Linux 2.6 提供如下关于等待队列的操作。

1. 定义“等待队列头”

```
wait_queue_head_t my_queue;
```

2. 初始化“等待队列头”

```
init_waitqueue_head(&my_queue);
```

而下面的 DECLARE_WAIT_QUEUE_HEAD() 宏可以作为定义并初始化等待队列头的“快捷方式”。

```
DECLARE_WAIT_QUEUE_HEAD (name)
```

3. 定义等待队列

```
DECLARE_WAITQUEUE (name, tsk)
```

该宏用于定义并初始化一个名为 name 的等待队列。

4. 添加/移除等待队列

```
void fastcall add_wait_queue(wait_queue_head_t *q, wait_queue_t *wait);
void fastcall remove_wait_queue(wait_queue_head_t *q, wait_queue_t *wait);
```

add_wait_queue() 用于将等待队列 wait 添加到等待队列头 q 指向的等待队列链表中，而 remove_wait_queue() 用于将等待队列 wait 从附属的等待队列头 q 指向的等待队列链表中移除。

5. 等待事件

```
wait_event(queue, condition)
wait_event_interruptible(queue, condition)
wait_event_timeout(queue, condition, timeout)
wait_event_interruptible_timeout(queue, condition, timeout)
```

等待第 1 个参数 queue 作为等待队列头的等待队列被唤醒，而且第 2 个参数 condition 必须满足，否则继续阻塞。wait_event() 和 wait_event_interruptible() 的区别在于后者可以被信号打断，而前者不能。加上_timeout 后的宏意味着阻塞等待的超时时间，以 jiffy 为单位，在第 3 个参数的 timeout 到达时，不论 condition 是否满足，均返回。

wait() 的定义如代码清单 8.3 所示，从其源代码可以看出，当 condition 满足时，wait_event() 会立即返回，否则，阻塞等待 condition 满足。

代码清单 8.3 wait_event() 函数

```
1 #define wait_event(wq, condition) \
2 do { \
3     if (condition) /*条件满足立即返回*/ \
4         break; \
5     __wait_event(wq, condition); /*添加等待队列并阻塞*/ \
6 } while (0)
7
8 #define __wait_event(wq, condition) \
9 do { \
10     DEFINE_WAIT(__wait); \
11 \
12     for (;;) { \
13         prepare_to_wait(&wq, &__wait, TASK_UNINTERRUPTIBLE); \
```



```
14     if (condition)                \
15         break;                    \
16     schedule(); /*放弃 CPU*/      \
17 }                                  \
18 finish_wait(&wq, &__wait);        \
19 } while (0)
20
21 void
22 prepare_to_wait(wait_queue_head_t *q, wait_queue_t *wait, int state)
23 {
24     unsigned long flags;
25
26     wait->flags &= ~WQ_FLAG_EXCLUSIVE;
27     spin_lock_irqsave(&q->lock, flags);
28     if (list_empty(&wait->task_list))
29         __add_wait_queue(q, wait); /* 加入等待队列 */
30     set_current_state(state); /* 设置进程状态 */
31     spin_unlock_irqrestore(&q->lock, flags);
32 }
33
34 void finish_wait(wait_queue_head_t *q, wait_queue_t *wait)
35 {
36     unsigned long flags;
37
38     __set_current_state(TASK_RUNNING); /* 恢复进程状态为 TASK_RUNNING */
39
40     if (!list_empty_careful(&wait->task_list)) {
41         spin_lock_irqsave(&q->lock, flags);
42         list_del_init(&wait->task_list);
43         spin_unlock_irqrestore(&q->lock, flags);
44     }
45 }
```

6. 唤醒队列

```
void wake_up(wait_queue_head_t *queue);
void wake_up_interruptible(wait_queue_head_t *queue);
```

上述操作会唤醒以 `queue` 作为等待队列头的所有等待队列中所有属于该等待队列头的等待队列对应的进程。

`wake_up()` 应该与 `wait_event()` 或 `wait_event_timeout()` 成对使用, 而 `wake_up_interruptible()` 则应与 `wait_event_interruptible()` 或 `wait_event_interruptible_timeout()` 成对使用。`wake_up()` 可唤醒处于 `TASK_INTERRUPTIBLE` 和 `TASK_UNINTERRUPTIBLE` 的进程, 而 `wake_up_interruptible()` 只能唤醒处于 `TASK_INTERRUPTIBLE` 的进程。

7. 在等待队列上睡眠

```
sleep_on(wait_queue_head_t *q);
interruptible_sleep_on(wait_queue_head_t *q);
```

`sleep_on()` 函数的作用就是将目前进程的状态置成 `TASK_UNINTERRUPTIBLE`, 并定义一个等待队列, 之后把它附属到等待队列头 `q`, 直到资源可获得, `q` 引导的等待队列被唤醒。

`interruptible_sleep_on()` 与 `sleep_on()` 函数类似, 其作用是将目前进程的状态置成 `TASK_INTERRUPTIBLE`, 并定义一个等待队列, 之后把它附属到等待队列头 `q`, 直到资源可获得, `q` 引导的等待队列被唤醒或者进程收到信号。

`sleep_on()`函数应该与 `wake_up()`成对使用, `interruptible_sleep_on()`应该与 `wake_up_interruptible()`成对使用。

代码清单 8.4 和 8.5 分别列出了 `sleep_on()`和 `interruptible_sleep_on()`函数的源代码。

代码清单 8.4 `sleep_on()`函数

```
1 void __sched sleep_on(wait_queue_head_t *q)
2 {
3     sleep_on_common(q, TASK_UNINTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
4 }
5
6 static long __sched
7 sleep_on_common(wait_queue_head_t *q, int state, long timeout)
8 {
9     unsigned long flags;
10    wait_queue_t wait;
11
12    init_waitqueue_entry(&wait, current);
13
14    __set_current_state(state);
15
16    spin_lock_irqsave(&q->lock, flags);
17    __add_wait_queue(q, &wait); /* 加入等待队列 */
18    spin_unlock(&q->lock);
19    timeout = schedule_timeout(timeout); /* 进程切换 */
20    spin_lock_irq(&q->lock);
21    __remove_wait_queue(q, &wait); /* 移除等待队列 */
22    spin_unlock_irqrestore(&q->lock, flags);
23
24    return timeout;
25 }
```

代码清单 8.5 `interruptible_sleep_on()`函数

```
1 void __sched interruptible_sleep_on(wait_queue_head_t *q)
2 {
3     sleep_on_common(q, TASK_INTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
4 }
```

从代码清单 8.4 和 8.5 可以看出, 不论是 `sleep_on()`还是 `interruptible_sleep_on()`, 都会调用 `sleep_on_common()`, 其流程如下。

(1) 定义并初始化一个等待队列, 将进程状态改变为 `TASK_UNINTERRUPTIBLE` (不能被信号打断) 或 `TASK_INTERRUPTIBLE` (可以被信号打断), 并将等待队列添加到等待队列头。

(2) 通过 `schedule_timeout()`放弃 CPU (这两个函数传递的超时参数都是 `MAX_SCHEDULE_TIMEOUT`, 即不会发生超时), 调度其他进程执行。

(3) 进程被其他地方唤醒, 将等待队列移出等待队列头。

在内核中使用 `set_current_state()`函数或 `__add_current_state()`函数来实现目前进程状态的改变, 直接采用 `current->state = TASK_UNINTERRUPTIBLE` 类似的赋值语句也是可行的。通常而言, `set_current_state()`函数在任何环境下都可以使用, 不会存在并发问题, 但是效率要低于 `__add_current_state()`。

因此, 在许多设备驱动中, 并不调用 `sleep_on()`或 `interruptible_sleep_on()`, 而是亲自进行进程



的状态改变和切换, 如代码清单 8.6 所示。

代码清单 8.6 在驱动程序中改变进程状态并调用 `schedule()`

```
1 static ssize_t xxx_write(struct file *file, const char *buffer, size_t count,
2     loff_t *ppos)
3 {
4     ...
5     DECLARE_WAITQUEUE(wait, current); /* 定义等待队列 */
6     add_wait_queue(&xxx_wait, &wait); /* 添加等待队列 */
7
8     ret = count;
9     /* 等待设备缓冲区可写 */
10    do {
11        avail = device_writable(...);
12        if (avail < 0)
13            __set_current_state(TASK_INTERRUPTIBLE); /* 改变进程状态 */
14
15        if (avail < 0) {
16            if (file->f_flags & O_NONBLOCK) { /* 非阻塞 */
17                if (!ret)
18                    ret = -EAGAIN;
19                goto out;
20            }
21            schedule(); /* 调度其他进程执行
22            if (signal_pending(current)) { /* 如果是因为信号唤醒 */
23                if (!ret)
24                    ret = -ERESTARTSYS;
25                goto out;
26            }
27        }
28    } while (avail < 0);
29
30    /* 写设备缓冲区 */
31    device_write(...)
32    out:
33    remove_wait_queue(&xxx_wait, &wait); /* 将等待队列移出等待队列头 */
34    set_current_state(TASK_RUNNING); /* 设置进程状态为 TASK_RUNNING */
35    return ret;
36 }
```

读懂代码清单 8.6 对理解 Linux 进程状态切换非常重要, 所以提请读者反复阅读此段代码 (尤其注意其中黑体的部分), 直至完全领悟。几个要点如下。

① 如果是非阻塞访问 (`O_NONBLOCK` 被设置), 设备忙时, 直接返回 “-EAGAIN”。

② 对于阻塞访问, 会进行状态切换并显式通过 “`schedule()`” 调度其他进程执行;

③ 醒来的时候要注意, 由于调度出去的时候, 进程状态是 `TASK_INTERRUPTIBLE`, 即浅度睡眠, 因此唤醒它的有可能是信号, 因此, 我们首先通过 “`signal_pending(current)`” 了解是不是信号唤醒的, 如果是, 立即返回 “-ERESTARTSYS”。

8.1.2 支持阻塞操作的 `globalfifo` 设备驱动

现在我们给 `globalmem` 增加这样的约束: 把 `globalmem` 中的全局内存变成一个 FIFO, 只有当

FIFO 中有数据的时候（即有进程把数据写到这个 FIFO 而且没有被读进程读空），读进程才能把数据读出，而且读取后的数据会从 `globalmem` 的全局内存中被拿掉；只有当 FIFO 非满时（即还有一些空间未被写，或写满后被读进程从这个 FIFO 中读出了数据），写进程才能往这个 FIFO 中写入数据。

现在，将 `globalmem` 重命名为“`globalfifo`”，在 `globalfifo` 中，读 FIFO 将唤醒写 FIFO，而写 FIFO 也将唤醒读 FIFO。首先，需要修改设备结构体，在其中增加两个等待队列头，分别对应于读和写，如代码清单 8.7 所示。

代码清单 8.7 `globalfifo` 设备结构体

```
1 struct globalfifo_dev {
2     struct cdev cdev; /*cdev 结构体*/
3     unsigned int current_len; /*fifo 有效数据长度*/
4     unsigned char mem[GLOBALFIFO_SIZE]; /*全局内存*/
5     struct semaphore sem; /*并发控制用的信号量*/
6     wait_queue_head_t r_wait; /*阻塞读用的等待队列头*/
7     wait_queue_head_t w_wait; /*阻塞写用的等待队列头*/
8 };
```

与 `globalfifo` 设备结构体的另一个不同是增加了 `current_len` 成员用于表征目前 FIFO 中有效数据的长度。

这个等待队列需在设备驱动模块加载函数中调用 `init_waitqueue_head()` 被初始化，新的设备驱动模块加载函数如代码清单 8.8 所示。

代码清单 8.8 `globalfifo` 设备驱动模块加载函数

```
1 int globalfifo_init(void)
2 {
3     int ret;
4     dev_t devno = MKDEV(globalfifo_major, 0);
5
6     /* 申请设备号*/
7     if (globalfifo_major)
8         ret = register_chrdev_region(devno, 1, "globalfifo");
9     else { /* 动态申请设备号 */
10         ret = alloc_chrdev_region(&devno, 0, 1, "globalfifo");
11         globalfifo_major = MAJOR(devno);
12     }
13     if (ret < 0)
14         return ret;
15     /* 动态申请设备结构体的内存*/
16     globalfifo_devp = kmalloc(sizeof(struct globalfifo_dev), GFP_KERNEL);
17     if (!globalfifo_devp) { /*申请失败*/
18         ret = - ENOMEM;
19         goto fail_malloc;
20     }
21
22     memset(globalfifo_devp, 0, sizeof(struct globalfifo_dev));
23
24     globalfifo_setup_cdev(globalfifo_devp, 0);
25 }
```



```
26  init_Mutex(&globalfifo_devp->sem); /*初始化信号量*/
27  init_waitqueue_head(&globalfifo_devp->r_wait); /*初始化读等待队列头*/
28  init_waitqueue_head(&globalfifo_devp->w_wait); /*初始化写等待队列头*/
29
30  return 0;
31
32  fail_malloc: unregister_chrdev_region(devno, 1);
33  return ret;
34 }
```

设备驱动读写操作需要被修改,在读函数中需增加等待 `globalfifo_devp->w_wait` 被唤醒的语句,而在写操作中唤醒 `globalfifo_devp->r_wait`,如代码清单 8.9 所示。

代码清单 8.9 增加等待队列后的 `globalfifo` 读写函数

```
1  /*globalfifo 读函数*/
2  static ssize_t globalfifo_read(struct file *filp, char __user *buf, size_t
3      count, loff_t *ppos)
4  {
5      int ret;
6      struct globalfifo_dev *dev = filp->private_data; /* 获得设备结构体指针
7          DECLARE_WAITQUEUE(wait, current); /* 定义等待队列
8
9      down(&dev->sem); /* 获得信号量
10     add_wait_queue(&dev->r_wait, &wait); /* 进入读等待队列头
11
12     /* 等待 FIFO 非空 */
13     while (dev->current_len == 0) {
14         if (filp->f_flags & O_NONBLOCK) {
15             ret = - EAGAIN;
16             goto out;
17         }
18         __set_current_state(TASK_INTERRUPTIBLE); /* 改变进程状态为睡眠
19         up(&dev->sem);
20
21         schedule(); /* 调度其他进程执行
22         if (signal_pending(current)) { /* 如果是因为信号唤醒 */
23             ret = - ERESTARTSYS;
24             goto out2;
25         }
26
27         down(&dev->sem);
28     }
29
30     /* 拷贝到用户空间 */
31     if (count > dev->current_len)
32         count = dev->current_len;
33
34     if (copy_to_user(buf, dev->mem, count)) {
35         ret = - EFAULT;
36         goto out;
37     } else {
38         memcpy(dev->mem, dev->mem + count, dev->current_len - count); /* fifo 数据前移*/
39         dev->current_len -= count; /* 有效数据长度减少
```

```

40     printk(KERN_INFO "read %d bytes(s),current_len:%d\n", count, dev
41         ->current_len);
42
43     wake_up_interruptible(&dev->w_wait); /* 唤醒写等待队列*/
44
45     ret = count;
46 }
47 out: up(&dev->sem); /* 释放信号量
48 out2: remove_wait_queue(&dev->w_wait, &wait); /* 移除等待队列*/
49 set_current_state(TASK_RUNNING);
50 return ret;
51 }
52
53
54 /*globalfifo 写操作*/
55 static ssize_t globalfifo_write(struct file *filp, const char __user *buf,
56     size_t count, loff_t *ppos)
57 {
58     struct globalfifo_dev *dev = filp->private_data; /* 获得设备结构体指针*/
59     int ret;
60     DECLARE_WAITQUEUE(wait, current); /* 定义等待队列*/
61
62     down(&dev->sem); /* 获取信号量*/
63     add_wait_queue(&dev->w_wait, &wait); /* 进入写等待队列头*/
64
65     /* 等待 FIFO 非满 */
66     while (dev->current_len == GLOBALFIFO_SIZE) {
67         if (filp->f_flags & O_NONBLOCK) {
68             /* 如果是非阻塞访问*/
69             ret = - EAGAIN;
70             goto out;
71         }
72         __set_current_state(TASK_INTERRUPTIBLE); /* 改变进程状态为睡眠*/
73         up(&dev->sem);
74
75         schedule(); /* 调度其他进程执行*/
76         if (signal_pending(current)) {
77             /* 如果是因为信号唤醒*/
78             ret = - ERESTARTSYS;
79             goto out2;
80         }
81
82         down(&dev->sem); /* 获得信号量 */
83     }
84
85     /*从用户空间拷贝到内核空间*/
86     if (count > GLOBALFIFO_SIZE - dev->current_len)
87         count = GLOBALFIFO_SIZE - dev->current_len;
88
89     if (copy_from_user(dev->mem + dev->current_len, buf, count)) {
90         ret = - EFAULT;
91         goto out;
92     } else {

```



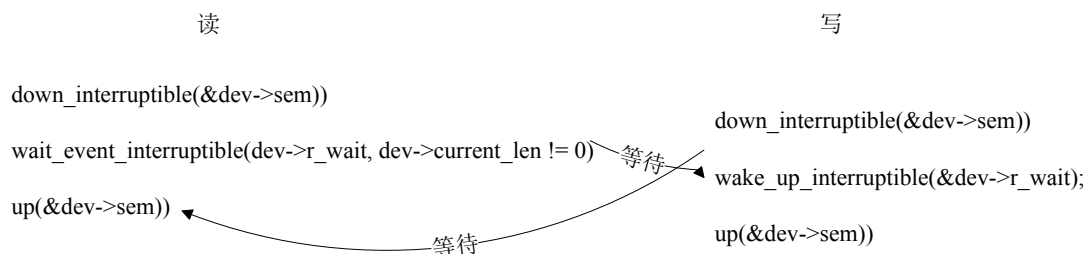
```
93     dev->current_len += count;
94     printk(KERN_INFO "written %d bytes(s),current_len:%d\n", count, dev
95         ->current_len);
96
97     wake_up_interruptible(&dev->r_wait); /* 唤醒读等待队列 */
98
99     ret = count;
100 }
101
102 out: up(&dev->sem); /* 释放信号量 */
103 out2: remove_wait_queue(&dev->w_wait, &wait);
104 set_current_state(TASK_RUNNING);
105 return ret;
106 }
```

在代码清单 8.9 中, 亲自掌管了等待队列进出和进程切换的过程, 现在会有一个疑问, 是否可以把读函数中一大段用于等待 `dev->current_len != 0` 的内容直接用 `wait_event_interruptible(dev->r_wait, dev->current_len != 0)` 替换, 把写函数中一大段用于等待 `dev->current_len != GLOBALFIFO_SIZE` 的代码用 `wait_event_interruptible(dev->w_wait, dev->current_len != 0)` 替换呢?

实际上, 就控制等待队列非空和非满的角度而言, `wait_event_interruptible(dev->r_wait, dev->current_len != 0)` 和第 13~28 行代码的功能完全一样, `wait_event_interruptible(dev->w_wait, dev->current_len != 0)` 和第 66~83 行代码的功能完全一样。细微的区别体现在第 13~28 行代码和第 66~83 行代码在进行 `schedule()` 即切换进程前, 通过 `up(&dev->sem)` 释放了信号量。这一细微的动作意义重大, 非如此, 则死锁将不可避免。

如图 8.1 (a) 所示, 假设目前的 FIFO 为空即 `dev->current_len` 为 0, 此时如果有一个读进程, 它会先获得信号量, 因为条件不满足, 它将因为 `wait_event_interruptible(dev->r_wait, dev->current_len != 0)` 而阻塞, 而释放 `dev->r_wait` 等待队列及让 `dev->current_len != 0` 的操作又需要在写进程中进行, 写进程在执行写操作前又必须等待读进程释放信号量, 造成互相等待对方资源的矛盾局面, 从而死锁。

如图 8.1 (b) 所示, 假设目前的 FIFO 为满即 `dev->current_len` 为 `GLOBALFIFO_SIZE`, 此时如果有一个写进程, 它先获得了信号量, 因为条件不满足, 它将因为 `wait_event_interruptible(dev->w_wait, dev->current_len != GLOBALFIFO_SIZE)` 而阻塞, 而释放 `dev->w_wait` 等待队列及让 `dev->current_len != GLOBALFIFO_SIZE` 的操作又需要在读进程中进行, 读进程在执行读操作前又必须等待写进程释放信号量, 造成互相等待对方资源的矛盾局面, 从而死锁。



(a) 队列为空时的死锁

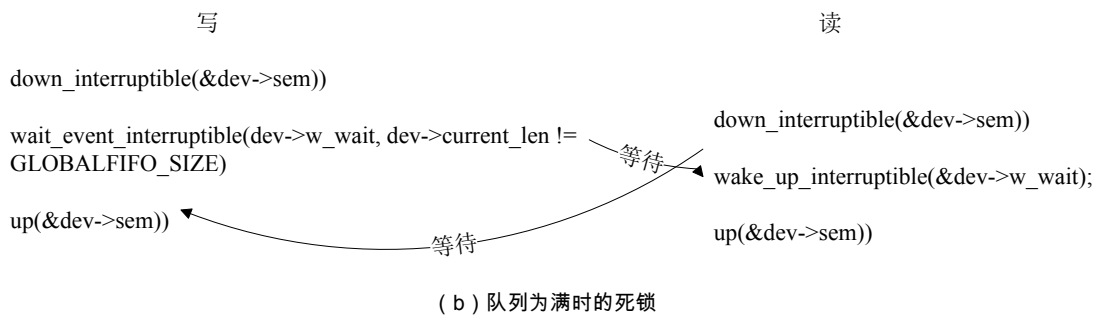


图 8.1 等待队列、信号量等引起的死锁

所谓死锁，就是多个进程循环等待它方占有的资源而无限期地僵持下去的局面。如果没有外力的作用，那么死锁涉及的各个进程都将永远处于封锁状态。因此，驱动工程师一定要注意：当多个等待队列、信号量等机制同时出现时，谨防死锁！

现在回过来看一下代码清单 8.9 的第 15 行和 75 行，发现在设备驱动的 `read()`、`write()` 等功能函数中，可以通过 `filp->f_flags` 标志获得用户空间是否要求非阻塞访问。驱动中可以依据此标志判断用户究竟要求阻塞还是非阻塞访问，从而进行不同的处理。

8.1.3 在用户空间验证 globalfifo 的读写

`/home/lihacker/develop/svn/ldd6410-read-only/training/kernel/drivers/globalfifo/ch8` 包含了 `globalfifo` 的驱动，运行“make”命令编译得到 `globalfifo.ko`。接着 `insmod` 模块：

```
lihacker@lihacker-laptop:~/develop/svn/ldd6410-read-only/training/kernel/drivers/globalfifo/ch8$ sudo su

root@lihacker-laptop:/home/lihacker/develop/svn/ldd6410-read-only/training/kernel/drivers/globalfifo/ch8# insmod globalfifo.ko
```

创建设备文件节点“`/dev/globalfifo`”：

```
root@lihacker-laptop:/home/lihacker/develop/svn/ldd6410-read-only/training/kernel/drivers/globalfifo/ch8# mknod /dev/globalfifo c 249 0
```

启动两个进程，一个进程“`cat /dev/globalfifo &`”在后台执行，一个进程“`echo 字符串 /dev/globalfifo`”在前台执行：

```
root@lihacker-laptop:/home/lihacker/develop/svn/ldd6410-read-only/training/kernel/drivers/globalfifo/ch8# cat /dev/globalfifo &
[1] 20910

root@lihacker-laptop:/home/lihacker/develop/svn/ldd6410-read-only/training/kernel/drivers/globalfifo/ch8# echo 'I want to be' > /dev/globalfifo
I want to be

root@lihacker-laptop:/home/lihacker/develop/svn/ldd6410-read-only/training/kernel/drivers/globalfifo/ch8# echo 'a great Chinese Linux driver Engineer' > /dev/globalfifo
a great Chinese Linux driver Engineer
```

每当 `echo` 进程向 `/dev/globalfifo` 写入一串数据，`cat` 进程就立即将该串数据显现出来，好的，让我们抱着这个信念“`I want to be a great Chinese Linux driver Engineer`”继续前行吧！



8.2 轮询操作

8.2.1 轮询的概念与作用

在用户程序中, `select()` 和 `poll()` 也是与设备阻塞与非阻塞访问息息相关的论题。使用非阻塞 I/O 的应用程序通常会使用 `select()` 和 `poll()` 系统调用查询是否可对设备进行无阻塞的访问。`select()` 和 `poll()` 系统调用最终会引发设备驱动中的 `poll()` 函数被执行, 在 2.5.45 内核中还引入了 `epoll()`, 即扩展的 `poll()`。

`select()` 和 `poll()` 系统调用的本质一样, 前者在 BSD UNIX 中引入, 后者在 System V 中引入。

8.2.2 应用程序中的轮询编程

应用程序中最广泛用到的是 BSD UNIX 中引入的 `select()` 系统调用, 其原型为:

```
int select(int numfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct
timeval *timeout);
```

其中 `readfds`、`writefds`、`exceptfds` 分别是被 `select()` 监视的读、写和异常处理的文件描述符集合, `numfds` 的值是需要检查的号码最高的文件描述符加 1。`timeout` 参数是一个指向 `struct timeval` 类型的指针, 它可以使 `select()` 在等待 `timeout` 时间后若没有文件描述符准备好则返回。`struct timeval` 数据结构的定义如代码清单 8.10 所示。

代码清单 8.10 `timeval` 结构体定义

```
1 struct timeval {
2     int tv_sec; /* 秒 */
3     int tv_usec; /* 微秒 */
4 };
```

下列操作用来设置、清除、判断文件描述符集合:

```
FD_ZERO(fd_set *set)
```

清除一个文件描述符集;

```
FD_SET(int fd, fd_set *set)
```

将一个文件描述符加入文件描述符集中;

```
FD_CLR(int fd, fd_set *set)
```

将一个文件描述符从文件描述符集中清除;

```
FD_ISSET(int fd, fd_set *set)
```

判断文件描述符是否被置位。

8.2.3 设备驱动中的轮询编程

设备驱动中 `poll()` 函数的原型是:

```
unsigned int(*poll)(struct file * filp, struct poll_table* wait);
```

第 1 个参数为 `file` 结构体指针, 第 2 个参数为轮询表指针。这个函数应该进行两项工作。

(1) 对可能引起设备文件状态变化的等待队列调用 `poll_wait()` 函数, 将对应的等待队列头添加到 `poll_table`。

(2) 返回表示是否能对设备进行无阻塞读、写访问的掩码。

关键的用于向 poll_table 注册等待队列的 poll_wait() 函数的原型如下：

```
void poll_wait(struct file *filp, wait_queue_head_t *queue, poll_table * wait);
```

poll_wait() 函数的名称非常容易让人产生误会，以为它和 wait_event() 等一样，会阻塞地等待某事件的发生，其实这个函数并不会引起阻塞。poll_wait() 函数所做的工作是把当前进程添加到 wait 参数指定的等待列表 (poll_table) 中。

驱动程序 poll() 函数应该返回设备资源的可获取状态，即 POLLIN、POLLOUT、POLLPRI、POLLERR、POLLNVAL 等宏的位“或”结果。每个宏的含义都表明设备的一种状态，如 POLLIN (定义为 0x0001) 意味着设备可以无阻塞地读，POLLOUT (定义为 0x0004) 意味着设备可以无阻塞地写。

通过以上分析，可得出设备驱动中 poll() 函数的典型模板，如代码清单 8.11 所示。

代码清单 8.11 poll() 函数典型模板

```
1 static unsigned int xxx_poll(struct file *filp, poll_table *wait)
2 {
3     unsigned int mask = 0;
4     struct xxx_dev *dev = filp->private_data; /*获得设备结构体指针*/
5
6     ...
7     poll_wait(filp, &dev->r_wait, wait); /* 加读等待队列头 */
8     poll_wait(filp, &dev->w_wait, wait); /* 加写等待队列头 */
9
10    if (...) /* 可读 */
11        mask |= POLLIN | POLLRDNORM; /*标示数据可获得*/
12
13    if (...) /* 可写 */
14        mask |= POLLOUT | POLLWRNORM; /*标示数据可写入*/
15    ...
16    return mask;
17 }
```

8.3 支持轮询操作的 globalfifo 驱动

8.3.1 在 globalfifo 驱动中增加轮询操作

在 globalfifo 的 poll() 函数中，首先将设备结构体中的 r_wait 和 w_wait 等待队列头添加到等待列表，然后通过判断 dev->current_len 是否等于 0 来获得设备的可读状态，通过判断 dev->current_len 是否等于 GLOBALFIFO_SIZE 来获得设备的可写状态，如代码清单 8.12 所示。

代码清单 8.12 globalfifo 设备驱动的 poll() 函数

```
1 static unsigned int globalfifo_poll(struct file *filp, poll_table *wait)
2 {
3     unsigned int mask = 0;
4     struct globalfifo_dev *dev = filp->private_data; /*获得设备结构体指针*/
5
```



```
6   down(&dev->sem);
7
8   poll_wait(filp, &dev->r_wait, wait);
9   poll_wait(filp, &dev->w_wait, wait);
10  /*fifo 非空*/
11  if (dev->current_len != 0)
12      mask |= POLLIN | POLLRDNORM; /*标示数据可获得*/
13  /*fifo 非满*/
14  if (dev->current_len != GLOBALFIFO_SIZE)
15      mask |= POLLOUT | POLLWRNORM; /*标示数据可写入*/
16
17  up(&dev->sem);
18  return mask;
19 }
```

注意, 要把 `globalfifo_poll` 赋给 `globalfifo_fops` 的 `poll` 成员:

```
static const struct file_operations globalfifo_fops = {
    ...
    .poll = globalfifo_poll,
    ...
};
```

8.3.2 在用户空间验证 globalfifo 设备的轮询

编写一个应用程序 `pollmonitor.c` 用于监控 `globalfifo` 的可读写状态, 这个程序如代码清单 8.13 所示。

代码清单 8.13 监控 `globalfifo` 是否可非阻塞读写的应用程序

```
1  #include ...
2
3  #define FIFO_CLEAR 0x1
4  #define BUFFER_LEN 20
5  main()
6  {
7      int fd, num;
8      char rd_ch[BUFFER_LEN];
9      fd_set rfds, wfds; /* 读/写文件描述符集
10
11      /*以非阻塞方式打开/dev/globalfifo 设备文件*/
12      fd = open("/dev/globalfifo", O_RDONLY | O_NONBLOCK);
13      if (fd != -1) {
14          /*FIFO 清 0*/
15          if (ioctl(fd, FIFO_CLEAR, 0) < 0)
16              printf("ioctl command failed\n");
17
18          while (1) {
19              FD_ZERO(&rfds);
20              FD_ZERO(&wfds);
21              FD_SET(fd, &rfds);
22              FD_SET(fd, &wfds);
23
24              select(fd + 1, &rfds, &wfds, NULL, NULL);
25              /*数据可获得*/
26              if (FD_ISSET(fd, &rfds))
```

```
27     printf("Poll monitor:can be read\n");
28     /*数据可写入*/
29     if (FD_ISSET(fd, &wfds))
30         printf("Poll monitor:can be written\n");
31     }
32 } else {
33     printf("Device open failure\n");
34 }
35 }
```

运行时看到，到没有任何输入，即 FIFO 为空时，程序不断地输出“Poll monitor:can be written”，当通过 echo 向/dev/globalfifo 写入一些数据后，将输出“Poll monitor:can be read”和“Poll monitor:can be written”，如果不断地通过 echo 向/dev/globalfifo 写入数据直至写满 FIFO，发现 pollmonitor 程序将只输出“Poll monitor:can be read”。对于 globalfifo 而言，不会出现既不能读、又不能写的情况。

8.4 总结

阻塞与非阻塞访问是 I/O 操作的两种不同模式，前者在 I/O 操作暂时不可进行时会让进程睡眠，后者则不然。

在设备驱动中阻塞 I/O 一般基于等待队列来实现，等待队列可用于同步驱动中事件发生的先后顺序。使用非阻塞 I/O 的应用程序也可借助轮询函数来查询设备是否能立即被访问，用户空间调用 select()和 poll()接口，设备驱动提供 poll()函数。设备驱动的 poll()本身不会阻塞，但是 poll()和 select()系统调用则会阻塞地等待文件描述符集合中的至少一个可访问或超时。

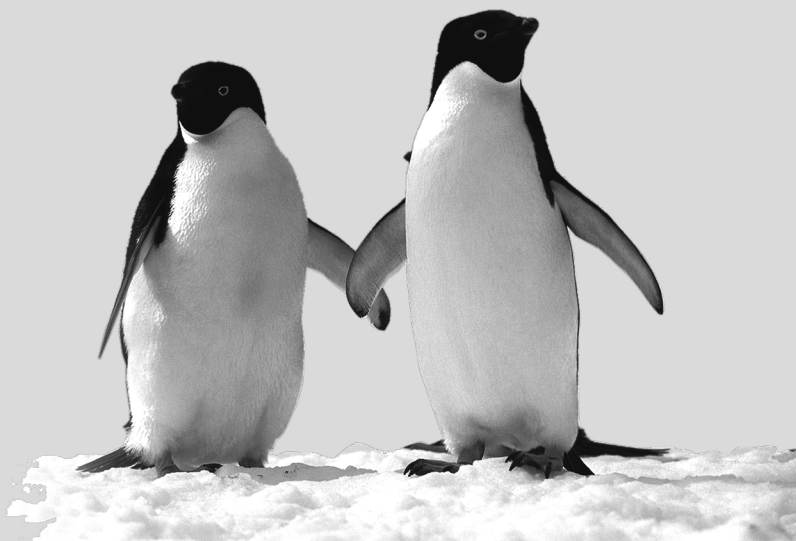
LINUX

第9章 Linux 设备驱动中的异步通知与异步 I/O

本章导读

在设备驱动中使用异步通知可以使得对设备的访问可进行时，由驱动主动通知应用程序进行访问。这样，使用无阻塞 I/O 的应用程序无需轮询设备是否可访问，而阻塞访问也可以被类似“中断”的异步通知所取代。

9.1 节讲解了异步通知的概念和作用，9.2 节讲解了 Linux 异步通知的编程方法，9.3 节给出了增加异步通知的 `globalfifo` 驱动及其在用户空间的验证。



9.1 异步通知的概念与作用

阻塞与非阻塞访问、poll()函数提供了较好地解决设备访问的机制，但是如果有了异步通知整套机制就更加完整了。

异步通知的意思是：一旦设备就绪，则主动通知应用程序，这样应用程序根本就不需要查询设备状态，这一点非常类似于硬件上“中断”的概念，比较准确的称谓是“信号驱动的异步 I/O”。信号是在软件层次上对中断机制的一种模拟，在原理上，一个进程收到一个信号与处理器收到一个中断请求可以说是一样的。信号是异步的，一个进程不必通过任何操作来等待信号的到达，事实上，进程也不知道信号到底什么时候到达。

阻塞 I/O 意味着一直等待设备可访问后再访问，非阻塞 I/O 中使用 poll()意味着查询设备是否可访问，而异步通知则意味着设备通知自身可访问，实现了异步 I/O。由此可见，这几种方式 I/O 可以互为补充。

图 9.1 呈现了阻塞 I/O，结合 poll()的非阻塞 I/O 及异步通知在时间先后顺序上的不同。

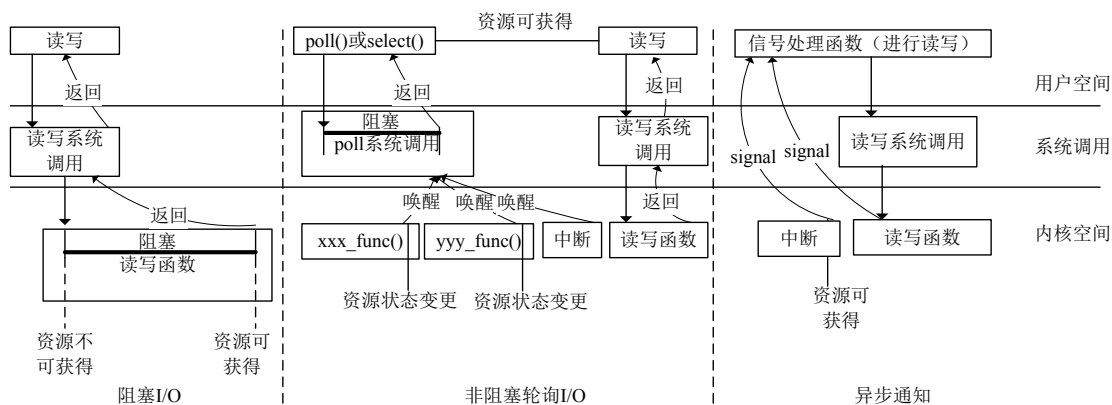


图 9.1 阻塞、非阻塞 I/O、异步通知区别

这里要强调的是，阻塞、非阻塞 I/O、异步通知本身没有优劣，应该根据不同的应用场景合理选择。

9.2 Linux 异步通知编程

9.2.1 Linux 信号

使用信号进行进程间通信（IPC）是 UNIX 中的一种传统机制，Linux 也支持这种机制。在 Linux 中，异步通知使用信号来实现，Linux 中可用的信号及其定义如表 9-1 所示。



表 9-1

Linux 信号

信 号	值	含 义
SIGHUP	1	挂起
SIGINT	2	终端中断
SIGQUIT	3	终端退出
SIGILL	4	无效命令
SIGTRAP	5	跟踪陷阱
SIGIOT	6	IOT 陷阱
SIGBUS	7	BUS 错误
SIGFPE	8	浮点异常
SIGKILL	9	强行终止 (不能被捕获或忽略)
SIGUSR1	10	用户定义的信号 1
SIGSEGV	11	无效的内存段处理
SIGUSR2	12	用户定义的信号 2
SIGPIPE	13	半关闭管道得写操作已经发生
SIGALRM	14	计时器到期
SIGTERM	15	终止
SIGSTKFLT	16	堆栈错误
SIGCHLD	17	子进程已经停止或退出
SIGCONT	18	如果停止了, 继续执行
SIGSTOP	19	停止执行 (不能被捕获或忽略)
SIGTSTP	20	终端停止信号
SIGTTIN	21	后台进程需要从终端读取输入
SIGTTOU	22	后台进程需要向从终端写出
SIGURG	23	紧急的套接字事件
SIGXCPU	24	超额使用 CPU 分配的时间
SIGXFSZ	25	文件尺寸超额
SIGVTALRM	26	虚拟时钟信号
SIGPROF	27	时钟信号描述
SIGWINCH	28	窗口尺寸变化
SIGIO	29	I/O
SIGPWR	30	断电重启

除了 SIGSTOP 和 SIGKILL 两个信号外, 进程能够忽略或捕获其他的全部信号。一个信号被捕获的意思是当一个信号到达时有相应的代码处理它。如果一个信号没有被这个进程所捕获, 内核将采用默认行为处理。

9.2.2 信号的接收

在用户程序中，为了捕获信号，可以使用 `signal()` 函数来设置对应信号的处理函数：

```
void (*signal(int signum, void (*handler)(int)))(int);
```

该函数原型较难理解，它可以分解为：

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

第一个参数指定信号的值，第二个参数指定针对前面信号值的处理函数，若为 `SIG_IGN`，表示忽略该信号；若为 `SIG_DFL`，表示采用系统默认方式处理信号；若为用户自定义的函数，则信号被捕获到后，该函数将被执行。

如果 `signal()` 调用成功，它返回最后一次为信号 `signum` 绑定的处理函数 `handler` 值，失败则返回 `SIG_ERR`。

在进程执行时，按下“Ctrl+c”将向其发出 `SIGINT` 信号，`kill` 正在运行的进程将向其发出 `SIGTERM` 信号，代码清单 9.1 的进程捕获这两个信号并输出信号值。

代码清单 9.1 `signal()` 捕获信号范例

```
1 void sigterm_handler(int signo)
2 {
3     printf("Have caught sig N.O. %d\n", signo);
4     exit(0);
5 }
6
7 int main(void)
8 {
9     signal(SIGINT, sigterm_handler);
10    signal(SIGTERM, sigterm_handler);
11    while(1);
12
13    return 0;
14 }
```

除了 `signal()` 函数外，`sigaction()` 函数可用于改变进程接收到特定信号后的行为，它的原型为：

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

该函数的第一个参数为信号的值，可以为除 `SIGKILL` 及 `SIGSTOP` 外的任何一个特定有效的信号。第二个参数是指向结构体 `sigaction` 的一个实例的指针，在结构体 `sigaction` 的实例中，指定了对特定信号的处理函数，若为空，则进程会以缺省方式对信号处理；第三个参数 `oldact` 指向的对象用来保存原来对相应信号的处理函数，可指定 `oldact` 为 `NULL`。如果把第二、第三个参数都设为 `NULL`，那么该函数可用于检查信号的有效性。

先来看一个使用信号实现异步通知的例子，它通过 `signal(SIGIO, input_handler)` 对标准输入文件描述符 `STDIN_FILENO` 启动信号机制。用户输入后，应用程序将接收到 `SIGIO` 信号，其处理函数 `input_handler()` 将被调用，如代码清单 9.2 所示。

代码清单 9.2 使用信号实现异步通知的应用程序实例

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <stdio.h>
4 #include <fcntl.h>
```



```
5 #include <signal.h>
6 #include <unistd.h>
7 #define MAX_LEN 100
8 void input_handler(int num)
9 {
10     char data[MAX_LEN];
11     int len;
12
13     /* 读取并输出 STDIN_FILENO 上的输入 */
14     len = read(STDIN_FILENO, &data, MAX_LEN);
15     data[len] = 0;
16     printf("input available:%s\n", data);
17 }
18
19 main()
20 {
21     int oflags;
22
23     /* 启动信号驱动机制 */
24     signal(SIGIO, input_handler);
25     fcntl(STDIN_FILENO, F_SETOWN, getpid());
26     oflags = fcntl(STDIN_FILENO, F_GETFL);
27     fcntl(STDIN_FILENO, F_SETFL, oflags | FASYNC);
28
29     /* 最后进入一个死循环, 仅为保持进程不终止, 如果程序中
30        没有这个死循环会立即执行完毕 */
31     while (1);
32 }
```

上述代码 24 行为 SIGIO 信号安装 input_handler() 作为处理函数, 第 25 行设置本进程为 STDIN_FILENO 文件的拥有者 (owner), 没有这一步内核不会知道应该将信号发给哪个进程。而为了启用异步通知机制, 还需对设备设置 FASYNC 标志, 26~27 行代码实现此目的。整个程序的执行效果如下:

```
[root@localhost driver_study]# ./signal_test
I am Chinese.
input available: I am Chinese.

I love Linux driver.
input available: I love Linux driver.
```

从中可以看出, 当用户输入一串字符后, 标准输入设备释放 SIGIO 信号, 这个信号“中断”驱使对应的应用程序中的 input_handler() 得以执行, 将用户输入显示出来。

由此可见, 为了在用户空间中能处理一个设备释放的信号, 它必须完成 3 项工作。

- (1) 通过 F_SETOWN IO 控制命令设置设备文件的拥有者为本进程, 这样从设备驱动发出的信号才能被本进程接收到。
- (2) 通过 F_SETFL IO 控制命令设置设备文件支持 FASYNC, 即异步通知模式。
- (3) 通过 signal() 函数连接信号和信号处理函数。

9.2.3 信号的释放

在设备驱动和应用程序的异步通知交互中, 仅仅在应用程序端捕获信号是不够的, 因为信号没有的源头在设备驱动端。因此, 应该在合适的时机让设备驱动释放信号, 在设备驱动程序中增

加信号释放的相关代码。

为了使设备支持异步通知机制，驱动程序中涉及 3 项工作。

(1) 支持 F_SETOWN 命令，能在这个控制命令处理中设置 `filp->f_owner` 为对应进程 ID。不过此项工作已由内核完成，设备驱动无需处理。

(2) 支持 F_SETFL 命令的处理，每当 FASYNC 标志改变时，驱动程序中的 `fasync()` 函数将得以执行。因此，驱动中应该实现 `fasync()` 函数。

(3) 在设备资源可获得时，调用 `kill_fasync()` 函数激发相应的信号。

驱动中的上述 3 项工作和应用程序中的 3 项工作是一一对应的，图 9.2 所示为异步通知处理过程中用户空间和设备驱动的交互。

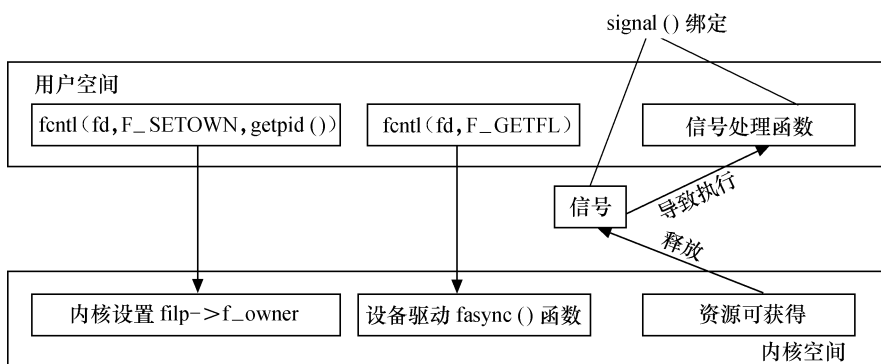


图 9.2 异步通知中设备驱动和异步通知的交互

设备驱动中异步通知编程比较简单，主要用到一项数据结构和两个函数。数据结构是 `fasync_struct` 结构体，两个函数分别是：

处理 FASYNC 标志变更的。

```
int fasync_helper(int fd, struct file *filp, int mode, struct fasync_struct **fa);
```

释放信号用的函数。

```
void kill_fasync(struct fasync_struct **fa, int sig, int band);
```

和其他的设备驱动一样，将 `fasync_struct` 结构体指针放在设备结构体中仍然是最佳选择，代码清单 9.3 给出了支持异步通知的设备结构体模板。

代码清单 9.3 支持异步通知的设备结构体模板

```
1 struct xxx_dev {
2     struct cdev cdev; /*cdev 结构体*/
3     ...
4     struct fasync_struct *async_queue; /* 异步结构体指针 */
5 };
```

在设备驱动的 `fasync()` 函数中，只需要简单地将该函数的 3 个参数以及 `fasync_struct` 结构体指针的指针作为第 4 个参数传入 `fasync_helper()` 函数即可。代码清单 9.4 给出了支持异步通知的设备驱动程序 `fasync()` 函数的模板。

代码清单 9.4 支持异步通知的设备驱动 `fasync()` 函数模板

```
1 static int xxx_fasync(int fd, struct file *filp, int mode)
2 {
```



```
3 struct xxx_dev *dev = filp->private_data;
4 return fasync_helper(fd, filp, mode, &dev->async_queue);
5 }
```

在设备资源可以获得时, 应该调用 `kill_fasync()` 释放 SIGIO 信号, 可读时第 3 个参数设置为 `POLL_IN`, 可写时第 3 个参数设置为 `POLL_OUT`。代码清单 9.5 给出了释放信号的范例。

代码清单 9.5 支持异步通知的设备驱动信号释放范例

```
1 static ssize_t xxx_write(struct file *filp, const char __user *buf, size_t count,
2                          loff_t *f_pos)
3 {
4     struct xxx_dev *dev = filp->private_data;
5     ...
6     /* 产生异步读信号 */
7     if (dev->async_queue)
8         kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
9     ...
10 }
```

最后, 在文件关闭时, 即在设备驱动的 `release()` 函数中, 应调用设备驱动的 `fasync()` 函数将文件从异步通知的列表中删除。代码清单 9.5 给出了支持异步通知的设备驱动 `release()` 函数的模板。

代码清单 9.6 支持异步通知的设备驱动 `release()` 函数模板

```
1 static int xxx_release(struct inode *inode, struct file *filp)
2 {
3     /* 将文件从异步通知列表中删除 */
4     xxx_fasync(-1, filp, 0);
5     ...
6     return 0;
7 }
```

9.3 支持异步通知的 globalfifo 驱动

9.3.1 在 globalfifo 驱动中增加异步通知

首先, 参照代码清单 9.2, 应该将异步结构体指针添加到 `globalfifo_dev` 设备结构体内, 如代码清单 9.7 所示。

代码清单 9.7 增加异步通知后的 globalfifo 设备结构体

```
1 struct globalfifo_dev {
2     struct cdev cdev; /*cdev 结构体*/
3     unsigned int current_len; /*fifo 有效数据长度*/
4     unsigned char mem[GLOBALFIFO_SIZE]; /*全局内存*/
5     struct semaphore sem; /*并发控制用的信号量*/
6     wait_queue_head_t r_wait; /*阻塞读用的等待队列头*/
7     wait_queue_head_t w_wait; /*阻塞写用的等待队列头*/
8     struct fasync_struct *async_queue; /* 异步结构体指针, 用于读 */
9 };
```

参考代码清单 9.3 的 `fasync()` 函数模板, `globalfifo` 的这个函数如代码清单 9.8。

代码清单 9.8 支持异步通知的 `globalfifo` 设备驱动 `fasync()` 函数

```
1 static int globalfifo_fasync(int fd, struct file *filp, int mode)
2 {
3     struct globalfifo_dev *dev = filp->private_data;
4     return fasync_helper(fd, filp, mode, &dev->async_queue);
5 }
```

在 `globalfifo` 设备被正确写入之后, 它变得可读, 这个时候驱动应释放 `SIGIO` 信号以便应用程序捕获, 代码清单 9.9 给出了支持异步通知的 `globalfifo` 设备驱动的写函数。

代码清单 9.9 支持异步通知的 `globalfifo` 设备驱动写函数

```
1 static ssize_t globalfifo_write(struct file *filp, const char __user *buf,
2     size_t count, loff_t *ppos)
3 {
4     struct globalfifo_dev *dev = filp->private_data; /* 获得设备结构体指针 */
5     int ret;
6     DECLARE_WAITQUEUE(wait, current); /* 定义等待队列 */
7
8     down(&dev->sem); /* 获取信号量 */
9     add_wait_queue(&dev->w_wait, &wait); /* 进入写等待队列头 */
10
11     /* 等待 FIFO 非满 */
12     if (dev->current_len == GLOBALFIFO_SIZE) {
13         if (filp->f_flags & O_NONBLOCK) { /* 如果是非阻塞访问 */
14             ret = -EAGAIN;
15             goto out;
16         }
17         __set_current_state(TASK_INTERRUPTIBLE); /* 改变进程状态为睡眠 */
18         up(&dev->sem);
19
20         schedule(); /* 调度其他进程执行 */
21         if (signal_pending(current)) { /* 如果是因为信号唤醒 */
22             ret = -ERESTARTSYS;
23             goto out2;
24         }
25
26         down(&dev->sem); /* 获得信号量 */
27     }
28
29     /* 从用户空间拷贝到内核空间 */
30     if (count > GLOBALFIFO_SIZE - dev->current_len)
31         count = GLOBALFIFO_SIZE - dev->current_len;
32
33     if (copy_from_user(dev->mem + dev->current_len, buf, count)) {
34         ret = -EFAULT;
35         goto out;
36     } else {
37         dev->current_len += count;
38         printk(KERN_INFO "written %d bytes(s), current_len: %d\n", count, dev
39             ->current_len);
40
41         wake_up_interruptible(&dev->r_wait); /* 唤醒读等待队列 */

```



```
42     /* 产生异步读信号 */
43     if (dev->async_queue)
44         kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
45
46     ret = count;
47 }
48
49 out: up(&dev->sem); /* 释放信号量 */
50 out2:remove_wait_queue(&dev->w_wait, &wait);
51 set_current_state(TASK_RUNNING);
52 return ret;
53 }
```

参考代码清单 9.6, 增加异步通知后的 globalfifo 设备驱动的 `release()` 函数中需调用 `globalfifo_fasync()` 函数将文件从异步通知列表中删除, 代码清单 9.10 给出了支持异步通知的 `globalfifo_release()` 函数。

代码清单 9.10 增加异步通知后的 globalfifo 设备驱动 `release()` 函数

```
1 int globalfifo_release(struct inode *inode, struct file *filp)
2 {
3     /* 将文件从异步通知列表中删除 */
4     globalfifo_fasync(-1, filp, 0);
5     return 0;
6 }
```

9.3.2 在用户空间验证 globalfifo 的异步通知

现在, 我们可以采用与代码清单 9.2 类似的方法, 编写一个在用户空间验证 globalfifo 异步通知的程序, 这个程序在接收到由 globalfifo 发出的信号后将输出信号值, 如代码清单 9.11 所示。

代码清单 9.11 监控 globalfifo 异步通知信号的应用程序

```
1 #include ...
2
3 /* 接收到异步读信号后的动作 */
4 void input_handler(int signum)
5 {
6     printf("receive a signal from globalfifo, signalnum:%d\n", signum);
7 }
8
9 main()
10 {
11     int fd, oflags;
12     fd = open("/dev/globalfifo", O_RDWR, S_IRUSR | S_IWUSR);
13     if (fd != -1) {
14         /* 启动信号驱动机制 */
15         signal(SIGIO, input_handler); /* 让 input_handler() 处理 SIGIO 信号 */
16         fcntl(fd, F_SETOWN, getpid());
17         oflags = fcntl(fd, F_GETFL);
18         fcntl(fd, F_SETFL, oflags | FASYNC);
19         while(1) {
20             sleep(100);
21         }
22     } else {
```

```

23     printf("device open failure\n");
24 }
25 }

```

/home/lihacker/develop/svn/ldd6410-read-only/training/kernel/drivers/globalfifo/ch9 包含了支持异步通知的 globalfifo 驱动以及代码清单 9.11 对应的 globalfifo_test.c 测试程序, 在该目录运行 make 将得到 globalfifo.ko 和 globalfifo_test:

```

lihacker@lihacker-laptop: ~ /develop/svn/ldd6410-read-only/training/kernel/drivers/globalfifo/ch9$ make
make -C /lib/modules/2.6.28-11-generic/build M=/home/lihacker/develop/svn/ldd6410-read-only/training/kernel/drivers/globalfifo/ch9 modules
make[1]: Entering directory `/usr/src/linux-headers-2.6.28-11-generic'
CC [M] /home/lihacker/develop/svn/ldd6410-read-only/training/kernel/drivers/globalfifo/ch9/globalfifo.o
Building modules, stage 2.
MODPOST 1 modules
CC /home/lihacker/develop/svn/ldd6410-read-only/training/kernel/drivers/globalfifo/ch9/globalfifo.mod.o
LD [M] /home/lihacker/develop/svn/ldd6410-read-only/training/kernel/drivers/globalfifo/ch9/globalfifo.ko
make[1]: Leaving directory '/usr/src/linux-headers-2.6.28-11-generic'
gcc -o globalfifo_test globalfifo_test.c

```

按照与 8.3.2 节相同的方法加载新的 globalfifo 设备驱动并创建设备文件节点, 运行上述程序, 每当通过 echo 向/dev/globalfifo 写入新的数据时, input_handler()将会被调用:

```

root@lihacker-laptop:/home/lihacker/develop/svn/ldd6410-read-only/training/kernel/drivers/globalfifo/ch9# ./globalfifo_test&
[1] 25251

root@lihacker-laptop:/home/lihacker/develop/svn/ldd6410-read-only/training/kernel/drivers/globalfifo/ch9# echo 1 > /dev/globalfifo
receive a signal from globalfifo,signalnum:29

root@lihacker-laptop:/home/lihacker/develop/svn/ldd6410-read-only/training/kernel/drivers/globalfifo/ch9# echo hello > /dev/globalfifo
receive a signal from globalfifo,signalnum:29

```

9.4 Linux 2.6 异步 I/O

9.4.1 AIO 概念与 GNU C 库函数

Linux 中最常用的输入/输出 (I/O) 模型是同步 I/O。在这个模型中, 当请求发出之后, 应用程序就会阻塞, 直到请求满足为止。这是很好的一种解决方案, 因为调用应用程序在等待 I/O 请求完成时不需要占用 CPU。但是在某些情况中, I/O 请求可能需要与其他进程产生交叠。可移植操作系统接口 (POSIX) 异步 I/O (AIO) 应用程序接口 (API) 就提供了这种功能。

Linux 异步 I/O 是 2.6 版本内核的一个标准特性, 但是我们在 2.4 版本内核的补丁中也可以找到它。AIO 基本思想是允许进程发起很多 I/O 操作, 而不用阻塞或等待任何操作完成。稍后或在接收到 I/O 操作完成的通知时, 进程再检索 I/O 操作的结果。



`select()` 函数所提供的功能（异步阻塞 I/O）与 AIO 类似，它对通知事件进行阻塞，而不是对 I/O 调用进行阻塞。

在异步非阻塞 I/O 中，我们可以同时发起多个传输操作。这需要每个传输操作都有惟一的上下文，这样才能在它们完成时区分到底是哪个传输操作完成了。在 AIO 中，通过 `aiocb`（AIO I/O Control Block）结构体进行区分。这个结构体包含了有关传输的所有信息，以及为数据准备的用户缓冲区。在产生 I/O 通知（称为完成）时，`aiocb` 结构就被用来惟一标识所完成的 I/O 操作。

AIO 系列 API 被 GNU C 库函数所包含，它被 POSIX.1b 所要求，主要包括如下函数。

1. `aio_read`

`aio_read()` 函数请求对一个有效的文件描述符进行异步读操作。这个文件描述符可以表示一个文件、套接字甚至管道。`aio_read` 函数的原型如下：

```
int aio_read( struct aiocb *aiocbp );
```

`aio_read()` 函数在请求进行排队之后会立即返回。如果执行成功，返回值就为 0；如果出现错误，返回值就为 -1，并设置 `errno` 的值。

2. `aio_write`

`aio_write()` 函数用来请求一个异步写操作。其函数原型如下：

```
int aio_write( struct aiocb *aiocbp );
```

`aio_write()` 函数会立即返回，说明请求已经进行排队（成功时返回值为 0，失败时返回值为 -1，并相应地设置 `errno`）。

3. `aio_error`

`aio_error()` 函数被用来确定请求的状态。其原型如下：

```
int aio_error( struct aiocb *aiocbp );
```

这个函数可以返回以下内容。

`EINPROGRESS`：说明请求尚未完成。

`ECANCELLED`：说明请求被应用程序取消了。

-1：说明发生了错误，具体错误原因由 `errno` 记录。

4. `aio_return`

异步 I/O 和标准 I/O 方式之间的另外一个区别是不能立即访问这个函数的返回状态，因为异步 I/O 并没有阻塞在 `read()` 调用上。在标准的 `read()` 调用中，返回状态是在该函数返回时提供的。但是在异步 I/O 中，我们要使用 `aio_return()` 函数。这个函数的原型如下：

```
ssize_t aio_return( struct aiocb *aiocbp );
```

只有在 `aio_error()` 调用确定请求已经完成（可能成功，也可能发生了错误）之后，才会调用这个函数。`aio_return()` 的返回值就等价于同步情况中 `read` 或 `write` 系统调用的返回值（所传输的字节数，如果发生错误，返回值为负数）。

代码清单 9.12 给出了用户空间应用程序进行异步读操作的一个例程，它首先打开文件，然后准备 `aiocb` 结构体，之后调用 `aio_read(&my_aiocb)` 进行提出异步读请求，当 `aio_error(&my_aiocb) == EINPROGRESS` 即操作还在进行中时，一直等待，结束后通过 `aio_return(&my_aiocb)` 获得返回值。

代码清单 9.12 用户空间异步读例程

```
1 #include <aio.h>
2 ...
```

```

3 int fd, ret;
4 struct aiocb my_aiocb;
5
6 fd = open("file.txt", O_RDONLY);
7 if (fd < 0)
8     perror("open");
9
10 /* 清零 aiocb 结构体 */
11 bzero((char*) &my_aiocb, sizeof(struct aiocb));
12
13 /* 为 aiocb 请求分配数据缓冲区 */
14 my_aiocb.aio_buf = malloc(BUFSIZE + 1);
15 if (!my_aiocb.aio_buf)
16     perror("malloc");
17
18 /* 初始化 aiocb 的成员 */
19 my_aiocb.aio_fildes = fd;
20 my_aiocb.aio_nbytes = BUFSIZE;
21 my_aiocb.aio_offset = 0;
22
23 ret = aio_read(&my_aiocb);
24 if (ret < 0)
25     perror("aio_read");
26
27 while (aio_error(&my_aiocb) == EINPROGRESS)
28     continue;
29
30 if ((ret = aio_return(&my_iocb)) > 0) {
31     /* 获得异步读的返回值 */
32 } else {
33     /* 读失败, 分析 errno */
34 }

```

5. aio_suspend

用户可以使用 `aio_suspend()` 函数来挂起（或阻塞）调用进程，直到异步请求完成为止，此时会产生一个信号，或者发生其他超时操作。调用者提供了一个 `aiocb` 引用列表，其中任何一个完成都会导致 `aio_suspend()` 返回。`aio_suspend` 的函数原型如下：

```

int aio_suspend( const struct aiocb *const cblist[],
                int n, const struct timespec *timeout );

```

代码清单 9.13 给出了用户空间异步读操作时使用 `aio_suspend()` 函数的例子。

代码清单 9.13 用户空间异步 I/O `aio_suspend()` 函数使用例程

```

1 struct aiocb *cblist[MAX_LIST]
2 /* 清零 aiocb 结构体链表 */
3 bzero( (char *)cblist, sizeof(cblist) );
4 /* 将一个或更多的 aiocb 放入 aiocb 结构体链表 */
5 cblist[0] = &my_aiocb;
6 ret = aio_read( &my_aiocb );
7 ret = aio_suspend( cblist, MAX_LIST, NULL );

```

6. aio_cancel

`aio_cancel()` 函数允许用户取消对某个文件描述符执行的一个或所有 I/O 请求。其原型如下：

```

int aio_cancel( int fd, struct aiocb *aiocbp );

```



要取消一个请求, 用户需提供文件描述符和 `aioctx` 指针。如果这个请求被成功取消了, 那么这个函数就会返回 `AIO_CANCELED`。如果请求完成了, 这个函数就会返回 `AIO_NOTCANCELED`。

要取消对某个给定文件描述符的所有请求, 用户需要提供这个文件的描述符, 并将 `aioctx` 参数设置为 `NULL`。如果所有的请求都取消了, 这个函数就会返回 `AIO_CANCELED`; 如果至少有一个请求没有被取消, 那么这个函数就会返回 `AIO_NOT_CANCELED`; 如果没有一个请求可以被取消, 那么这个函数就会返回 `AIO_ALLDONE`。然后, 可以使用 `aio_error()` 来验证每个 `AIO` 请求, 如果某请求已经被取消了, 那么 `aio_error()` 就会返回 `-1`, 并且 `errno` 会被设置为 `ECANCELED`。

7. `lio_listio`

`lio_listio()` 函数可用于同时发起多个传输。这个函数非常重要, 它使得用户可以在一个系统调用 (一次内核上下文切换) 中启动大量的 I/O 操作。`lio_listio` API 函数的原型如下:

```
int lio_listio( int mode, struct aiocb *list[], int nent, struct sigevent *sig );
```

`mode` 参数可以是 `LIO_WAIT` 或 `LIO_NOWAIT`。`LIO_WAIT` 会阻塞这个调用, 直到所有的 I/O 都完成为止。在操作进行排队之后, `LIO_NOWAIT` 就会返回。`list` 是一个 `aiocb` 引用的列表, 最大元素的个数是由 `nent` 定义的。如果 `list` 的元素为 `NULL`, `lio_listio()` 会将其忽略。

代码清单 9.14 给出了用户空间异步 I/O 操作时使用 `lio_listio()` 函数的例子。

代码清单 9.14 用户空间异步 I/O `lio_listio()` 函数使用例程

```
1 struct aiocb aiocb1, aiocb2;
2 struct aiocb *list[MAX_LIST];
3 ...
4 /* 准备第一个 aiocb */
5 aiocb1.aio_fildes = fd;
6 aiocb1.aio_buf = malloc( BUFSIZE+1 );
7 aiocb1.aio_nbytes = BUFSIZE;
8 aiocb1.aio_offset = next_offset;
9 aiocb1.aio_lio_opcode = LIO_READ; /*异步读操作*/
10 ... /*准备多个 aiocb */
11 bzero( (char *)list, sizeof(list) );
12
13 /*将 aiocb 填入链表*/
14 list[0] = &aiocb1;
15 list[1] = &aiocb2;
16 ...
17 ret = lio_listio( LIO_WAIT, list, MAX_LIST, NULL ); /*发起大量 I/O 操作*/
```

上述代码第 9 行中, 因为是进行异步读操作, 所以操作码为 `LIO_READ`, 对于写操作来说, 应该使用 `LIO_WRITE` 作为操作码, 而 `LIO_NOP` 意味着空操作。

网页 http://www.gnu.org/software/libc/manual/html_node/Asynchronous-I_002fo.html 包含了 `AIO` 库函数的详细信息。

9.4.2 使用信号作为 AIO 的通知

9.1~9.3 节讲述的信号作为异步通知的机制在 `AIO` 中仍然是适用的, 为使用信号, 使用 `AIO` 的应用程序同样需要定义信号处理程序, 在指定的信号被产生时会触发调用这个处理程序。作为信号上下文的一部分, 特定的 `aiocb` 请求被提供给信号处理函数用来区分 `AIO` 请求。

代码清单 9.15 给出了使用信号作为 `AIO` 异步 I/O 通知机制的例子。

代码清单 9.15 使用信号作为 AIO 异步 I/O 通知机制例程

```

1  /*设置异步 I/O 请求*/
2  void setup_io(...)
3  {
4      int fd;
5      struct sigaction sig_act;
6      struct aiocb my_aiocb;
7      ...
8      /* 设置信号处理函数 */
9      sigemptyset(&sig_act.sa_mask);
10     sig_act.sa_flags = SA_SIGINFO;
11     sig_act.sa_sigaction = aio_completion_handler;
12
13     /* 设置 AIO 请求 */
14     bzero((char*) &my_aiocb, sizeof(struct aiocb));
15     my_aiocb.aio_fildes = fd;
16     my_aiocb.aio_buf = malloc(BUF_SIZE + 1);
17     my_aiocb.aio_nbytes = BUF_SIZE;
18     my_aiocb.aio_offset = next_offset;
19
20     /* 连接 AIO 请求和信号处理函数 */
21     my_aiocb.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
22     my_aiocb.aio_sigevent.sigev_signo = SIGIO;
23     my_aiocb.aio_sigevent.sigev_value.sival_ptr = &my_aiocb;
24
25     /* 将信号与信号处理函数绑定 */
26     ret = sigaction(SIGIO, &sig_act, NULL);
27     ...
28     ret = aio_read(&my_aiocb); /*发出异步读请求*/
29 }
30
31 /*信号处理函数*/
32 void aio_completion_handler(int signo, siginfo_t *info, void *context)
33 {
34     struct aiocb *req;
35
36     /* 确定是我们需要的信号*/
37     if (info->si_signo == SIGIO) {
38         req = (struct aiocb*)info->si_value.sival_ptr; /*获得 aiocb*/
39
40         /* 请求的操作完成了吗? */
41         if (aio_error(req) == 0) {
42             /* 请求的操作完成, 获取返回值 */
43             ret = aio_return(req);
44         }
45     }
46 }

```

特别要注意上述代码的第 38 行通过(struct aiocb*)info->si_value.sival_ptr 获得了信号对应的 aiocb。

9.4.3 使用回调函数作为 AIO 的通知

除了信号之外, 应用程序还可提供一个回调 (Callback) 函数给内核, 以便 AIO 的请求完成



后内核调用这个函数。

代码清单 9.16 给出了使用回调函数作为 AIO 异步 I/O 请求完成的通知机制的例子。

代码清单 9.16 使用回调函数作为 AIO 异步 I/O 通知机制例程

```
1 /*设置异步 I/O 请求*/
2 void setup_io(...)
3 {
4     int fd;
5     struct aiocb my_aiocb;
6     ...
7     /* 设置 AIO 请求 */
8     bzero((char*) &my_aiocb, sizeof(struct aiocb));
9     my_aiocb.aio_fildes = fd;
10    my_aiocb.aio_buf = malloc(BUF_SIZE + 1);
11    my_aiocb.aio_nbytes = BUF_SIZE;
12    my_aiocb.aio_offset = next_offset;
13
14    /* 连接 AIO 请求和线程回调函数 */
15    my_aiocb.aio_sigevent.sigev_notify = SIGEV_THREAD;
16    my_aiocb.aio_sigevent.notify_function = aio_completion_handler;
17    /*设置回调函数*/
18    my_aiocb.aio_sigevent.notify_attributes = NULL;
19    my_aiocb.aio_sigevent.sigev_value.sival_ptr = &my_aiocb;
20    ...
21    ret = aio_read(&my_aiocb); /* 发起 AIO 请求*/
22 }
23
24 /* 异步 I/O 完成回调函数 */
25 void aio_completion_handler(sigval_t sigval)
26 {
27     struct aiocb *req;
28     req = (struct aiocb*)sigval.sival_ptr;
29
30     /* AIO 请求完成? */
31     if (aio_error(req) == 0)
32         /* 请求完成, 获得返回值 */
33         ret = aio_return(req);
34 }
```

上述程序在创建 aiocb 请求之后, 使用 SIGEV_THREAD 请求了一个线程回调函数来作为通知方法。在回调函数中, 通过(struct aiocb*)sigval.sival_ptr 可以获得对应的 aiocb 指针, 使用 AIO 函数可验证请求是否已经完成。

proc 文件系统包含了两个虚拟文件, 它们可以用来对异步 I/O 的性能进行优化。

(1) /proc/sys/fs/aio-nr 文件提供了系统范围异步 I/O 请求现在的数目。

(2) /proc/sys/fs/aio-max-nr 文件是所允许的并发请求的最大个数, 最大个数通常是 64KB, 这对于大部分应用程序来说都已经足够了。

9.4.4 AIO 与设备驱动

在内核中, 每个 I/O 请求都对应于一个 kiocb 结构体, 其 ki_filp 成员指向对应的 file 指针, 通过 is_sync_kiocb() 可以判断某 kiocb 是否为同步 I/O 请求, 如果返回非真, 表示为异步 I/O 请求。

块设备和网络设备本身是异步的，只有字符设备必须明确表明应支持 AIO。AIO 对于大多数字符设备而言都不是必须的，只有极少数设备需要。比如，对于磁带机，由于 I/O 操作很慢，这时候使用异步 I/O 将可改善性能。

字符设备驱动程序中，`file_operations` 包含 3 个与 AIO 相关的成员函数：

```
ssize_t (*aio_read) (struct kiocb *iocb, char *buffer,
size_t count, loff_t offset);
ssize_t (*aio_write) (struct kiocb *iocb, const char *buffer,
size_t count, loff_t offset);
int (*aio_fsync) (struct kiocb *iocb, int datasync);
```

`aio_read()` 和 `aio_write()` 与 `file_operations` 中的 `read()` 和 `write()` 中的 `offset` 参数不同，它直接传递值，而后者传递的是指针，这是因为 AIO 从来不需要改变文件的位置。

`aio_read()` 和 `aio_write()` 函数本身不一定完成了读和写操作，它只是发起、初始化读和写操作，代码清单 9.17 给出了驱动程序中 `aio_read()` 和 `aio_write()` 函数的实现例子。

代码清单 9.17 设备驱动中的异步 I/O 函数

```
1 struct async_work {
2     struct kiocb *iocb; /* kiocb 结构体指针*/
3     int result; /*执行结果 */
4     struct work_struct work; /* 工作结构体 */
5 };
6 ...
7 /*异步读*/
8 static ssize_t xxx_aio_read(struct kiocb *iocb, char *buf, size_t count, loff_t
9     pos)
10 {
11     return xxx_defer_op(0, iocb, buf, count, pos);
12 }
13
14 /*异步写*/
15 static ssize_t xxx_aio_write(struct kiocb *iocb, const char *buf, size_t count,
16     loff_t pos)
17 {
18     return xxx_defer_op(1, iocb, (char*)buf, count, pos);
19 }
20
21 /*初始化异步 I/O*/
22 static int xxx_defer_op(int write, struct kiocb *iocb, char *buf, size_t count,
23     loff_t pos)
24 {
25     struct async_work *async_wk;
26     int result;
27     /* 当我们能访问 buffer 时进行 copy */
28     if (write)
29         result = xxx_write(iocb->ki_filp, buf, count, &pos);
30     else
31         result = xxx_read(iocb->ki_filp, buf, count, &pos);
32     /* 如果是同步 IOCB，立即返回状态 */
33     if (is_sync_kiocb(iocb))
34         return result;
35
36     /* 否则，推后几微秒执行 */
```



```
37     async_wk = kmalloc(sizeof(*async_wk), GFP_KERNEL);
38     if (async_wk == NULL)
39         return result;
40     /*调度延迟的工作*/
41     async_wk->iocb = iocb;
42     async_wk->result = result;
43     INIT_WORK(&async_wk->work, xxx_do_deferred_op, async_wk);
44     schedule_delayed_work(&async_wk->work, Hz / 100);
45     return - EIOCBQUEUED; /*控制权返回用户空间*/
46 }
47
48 /*延迟后执行*/
49 static void xxx_do_deferred_op(void *p)
50 {
51     struct async_work *async_wk = (struct async_work*)p;
52     ... /* 执行 I/O 操作 */
53     aio_complete(async_wk->iocb, async_wk->result, 0);
54     kfree(async_wk);
55 }
```

上述代码中最核心的是使用 work_struct 机制通过 schedule_delayed_work() 函数将 I/O 操作延迟后执行，而在具体的 I/O 操作执行完成后，53 行调用 aio_complete() 通知内核驱动程序已经完成了 I/O 操作。

通常而言，具体的字符设备驱动一般不需要实现 AIO 支持，而内核中仅有 fs/direct-io.c，drivers/usb/gadget/inode.c、fs/nfs/direct.c 等少量地方使用了 AIO。

9.5 总结

本章主要讲述了 Linux 中的异步 I/O，异步 I/O 可以使得应用程序在等待 I/O 操作的同时进行其他操作。

使用信号可以实现设备驱动与用户程序之间的异步通知，总体而言，设备驱动和用户空间要分别完成 3 项对应的工作，用户空间设置文件的拥有者、FASYNC 标志及捕获信号，内核空间响应文件的拥有者、FASYNC 标志的设置并在资源可获得时释放信号。

Linux 2.6 内核包含对 AIO 的支持，它为用户空间提供了统一的异步 I/O 接口。在 AIO 中，信号和回调函数是实现内核空间对用户空间应用程序通知的两种机制。

LINUX

第10章

中断与时钟

本章导读

本章主要讲解 Linux 设备驱动编程中的中断与定时器处理。由于中断服务程序的执行并不存在于进程上下文，因此，要求中断服务程序的时间尽可能地短。因此，Linux 在中断处理中引入了顶半部和底半部分离的机制。另外，内核中对时钟的处理也采用中断方式，而内核软件定时器最终依赖于时钟中断。

10.1 节讲解中断和定时器的概念及处理流程。

10.2 节讲解 Linux 中断处理程序的架构，顶半部、底半部之间的关系。

10.3 节讲解 Linux 中断编程的方法，涉及申请和释放中断、禁止和使能中断以及中断底半部 tasklet、工作队列、软中断机制。

10.4 节讲解多个设备共享同一个中断号时的中断处理过程。

10.5 节和 10.6 节分别讲解 Linux 设备驱动编程中定时器的编程以及内核延时的方法。





10.1 中断与定时器

所谓中断是指 CPU 在执行程序的过程中, 出现了某些突发事件急待处理, CPU 必须暂停执行当前的程序, 转去处理突发事件, 处理完毕后 CPU 又返回原程序被中断的位置并继续执行。

根据中断的来源, 中断可分为内部中断和外部中断, 内部中断的中断源来自 CPU 内部 (软件中断指令、溢出、除法错误等, 例如, 操作系统从用户态切换到内核态需借助 CPU 内部的软件中断), 外部中断的中断源来自 CPU 外部, 由外设提出请求。

根据中断是否可以屏蔽分为可屏蔽中断与不屏蔽中断 (NMI), 可屏蔽中断可以通过屏蔽字被屏蔽, 屏蔽后, 该中断不再得到响应, 而不屏蔽中断不能被屏蔽。

根据中断入口跳转方法的不同, 分为向量中断和非向量中断。采用向量中断的 CPU 通常为不同的中断分配不同的中断号, 当检测到某中断号的中断到来后, 就自动跳转到与该中断号对应的地址执行。不同中断号的中断有不同的入口地址。非向量中断的多个中断共享一个入口地址, 进入该入口地址后再通过软件判断中断标志来识别具体是哪个中断。也就是说, 向量中断由硬件提供中断服务程序入口地址, 非向量中断由软件提供中断服务程序入口地址。

一个典型的非向量中断服务程序如代码清单 10.1 所示, 它先判断中断源, 然后调用不同中断源的中断服务程序。

代码清单 10.1 非向量中断服务程序典型结构

```
1  irq_handler()
2  {
3      ...
4      int int_src = read_int_status(); /*读硬件的中断相关寄存器*/
5      switch (int_src) { /*判断中断源*/
6          case DEV_A:
7              dev_a_handler();
8              break;
9          case DEV_B:
10             dev_b_handler();
11             break;
12             ...
13         default:
14             break;
15     }
16     ...
17 }
```

嵌入式系统以及 X86 PC 中大多包含可编程中断控制器 (PIC), 许多 MCU 内部就集成了 PIC。如在 80386 中, PIC 是两片 i8259A 芯片的级联。通过读写 PIC 的寄存器, 程序员可以屏蔽/使能某中断及获得中断状态, 前者一般通过中断 MASK 寄存器完成, 后者一般通过中断 PEND 寄存器完成。

定时器在硬件上也依赖中断来实现, 图 10.1 所示为典型的嵌入式微处理内可编程间隔定时器 (PIT) 的工作原理, 它接收一个时钟输入, 当时钟脉冲到来时, 将目前计数值增 1 并与

预先设置的计数值（计数目标）比较，若相等，证明计数周期满，产生定时器中断并复位目前计数值。

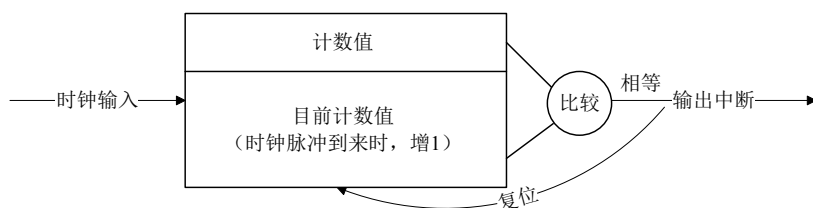


图 10.1 定时器工作原理

10.2 Linux 中断处理程序架构

设备的中断会打断内核中进程的正常调度和运行，系统对更高吞吐率的追求势必要求中断服务程序尽可能的短小精悍。但是，这个良好的愿望往往与现实并不吻合。在大多数真实的系统中，当中断到来时，要完成的工作往往并不会是短小的，它可能要进行较大的耗时处理。

图 10.2 描述了 Linux 内核的中断处理机制。为了在中断执行时间尽可能短和中断处理需完成大量工作之间找到一个平衡点，Linux 将中断处理程序分解为两个半部：顶半部（top half）和底半部（bottom half）。

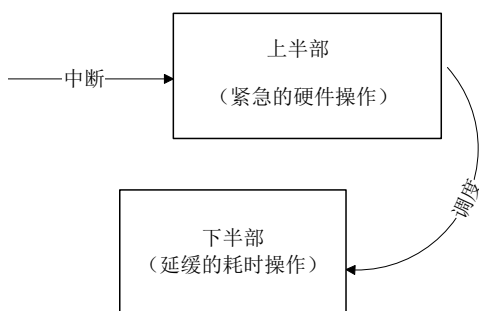


图 10.2 Linux 中断处理机制

顶半部完成尽可能少的比较紧急的功能，它往往只是简单地读取寄存器中的中断状态并清除中断标志后就进行“登记中断”的工作。“登记中断”意味着将底半部处理程序挂到该设备的底半部执行队列中去。这样，顶半部执行的速度就会很快，可以服务更多的中断请求。

现在，中断处理工作的重心就落在了底半部的头上，它来完成中断事件的绝大多数任务。底半部几乎做了中断处理程序所有的事情，而且可以被新的中断打断，这也是底半部和顶半部的最大不同，因为顶半部往往被设计成不可中断。底半部则相对来说并不是非常紧急的，而且相对比较耗时，不在硬件中断服务程序中执行。

尽管顶半部、底半部的结合能够改善系统的响应能力，但是，僵化地认为 Linux 设备驱动中的中断处理一定要分两个半部则是不对的。如果中断要处理的工作本身很少，则完全可以直接在顶半部全部完成。



其他操作系统中对中断的处理也采用了类似于 Linux 的方法，真正的硬件中断服务程序都应该尽可能短。因此，许多操作系统都提供了中断上下文和非中断上下文相结合的机制，将中断的耗时工作保留到非中断上下文去执行。例如，在 VxWorks 中，网络设备包接收中断到来后，中断服务程序会通过 netJobAdd() 函数将耗时的包接收和上传工作交给 tNetTask 任务去执行。



在 Linux 中, 查看 `/proc/interrupts` 文件可以获得系统中中断的统计信息, 在单处理器的系统中, 第 1 列是中断号, 第 2 列是向 CPU0 产生该中断的次数, 之后的是对于中断的描述。

```
Cpu0
0:      135253      XT-PIC  timer
1:         22      XT-PIC  i8042
2:          0      XT-PIC  cascade
8:          1      XT-PIC  rtc
10:       108      XT-PIC  eth0
11:      3707      XT-PIC  BusLogic BT-958
12:       313      XT-PIC  i8042
15:          4      XT-PIC  idel
NMI:          0
ERR:          0
```

10.3 Linux 中断编程

10.3.1 申请和释放中断

在 Linux 设备驱动中, 使用中断的设备需要申请和释放对应的中断, 分别使用内核提供的 `request_irq()` 和 `free_irq()` 函数。

1. 申请 IRQ

```
int request_irq(unsigned int irq, irq_handler_t handler,
               unsigned long irqflags, const char *devname, void *dev_id)
```

`irq` 是要申请的硬件中断号。

`handler` 是向系统登记的中断处理函数 (顶半部), 是一个回调函数, 中断发生时, 系统调用这个函数, `dev_id` 参数将被传递给它。

`irqflags` 是中断处理的属性, 可以指定中断的触发方式以及处理方式。在触发方式方面, 可以是 `IRQF_TRIGGER_RISING`、`IRQF_TRIGGER_FALLING`、`IRQF_TRIGGER_HIGH`、`IRQF_TRIGGER_LOW` 等。在处理方式方面, 若设置了 `IRQF_DISABLED`, 表明中断处理程序是快速处理程序, 快速处理程序被调用时屏蔽所有中断, 慢速处理程序则不会屏蔽其他设备的驱动; 若设置了 `IRQF_SHARED`, 则表示多个设备共享中断, `dev_id` 在中断共享时会用到, 一般设置为这个设备的设备结构体或者 `NULL`。

`request_irq()` 返回 0 表示成功, 返回 `-EINVAL` 表示中断号无效或处理函数指针为 `NULL`, 返回 `-EBUSY` 表示中断已经被占用且不能共享。

顶半部 `handler` 的类型 `irq_handler_t` 定义为:

```
typedef irqreturn_t (*irq_handler_t)(int, void *);
typedef int irqreturn_t;
```

2. 释放 IRQ

与 `request_irq()` 相对应的函数为 `free_irq()`, `free_irq()` 的原型为:

```
void free_irq(unsigned int irq, void *dev_id);
```

`free_irq()` 中参数的定义与 `request_irq()` 相同。

10.3.2 使能和屏蔽中断

下列 3 个函数用于屏蔽一个中断源：

```
void disable_irq(int irq);
void disable_irq_nosync(int irq);
void enable_irq(int irq);
```

`disable_irq_nosync()`与 `disable_irq()`的区别在于前者立即返回，而后者等待目前的中断处理完成。由于 `disable_irq()`会等待指定的中断被处理完，因此如果在 `n` 号中断的顶半部调用 `disable_irq(n)`，会引起系统的死锁，这种情况下，只能调用 `disable_irq_nosync(n)`。

下列两个函数（或宏，具体实现依赖于 CPU 体系结构）将屏蔽本 CPU 内的所有中断：

```
#define local_irq_save(flags) ...
void local_irq_disable(void);
```

前者会将目前的中断状态保留在 `flags` 中（注意 `flags` 为 `unsigned long` 类型，被直接传递，而不是通过指针），后者直接禁止中断而不保存状态。

与上述两个禁止中断对应的恢复中断的函数（或宏）是：

```
#define local_irq_restore(flags) ...
void local_irq_enable(void);
```

以上各 `local_` 开头的方法的作用范围是本 CPU 内。

10.3.3 底半部机制

Linux 实现底半部的机制主要有 `tasklet`、工作队列和软中断。

1. tasklet

`tasklet` 的使用较简单，我们只需要定义 `tasklet` 及其处理函数并将两者关联，例如：

```
void my_tasklet_func(unsigned long); /*定义一个处理函数*/
DECLARE_TASKLET(my_tasklet, my_tasklet_func, data);
/*定义一个 tasklet 结构 my_tasklet，与 my_tasklet_func(data) 函数相关联*/
```

代码 `DECLARE_TASKLET(my_tasklet, my_tasklet_func, data)` 实现了定义名称为 `my_tasklet` 的 `tasklet` 并将其与 `my_tasklet_func()` 这个函数绑定，而传入这个函数的参数为 `data`。

在需要调度 `tasklet` 的时候引用一个 `tasklet_schedule()` 函数就能使系统在适当的时候进行调度运行：

```
tasklet_schedule(&my_tasklet);
```

使用 `tasklet` 作为底半部处理中断的设备驱动程序模板如代码清单 10.2 所示（仅包含与中断相关的部分）。

代码清单 10.2 tasklet 使用模板

```
1 /*定义 tasklet 和底半部函数并关联*/
2 void xxx_do_tasklet(unsigned long);
3 DECLARE_TASKLET(xxx_tasklet, xxx_do_tasklet, 0);
4
5 /*中断处理底半部*/
6 void xxx_do_tasklet(unsigned long)
7 {
8     ...
9 }
10
```



```
11 /*中断处理顶半部*/
12 irqreturn_t xxx_interrupt(int irq, void *dev_id)
13 {
14     ...
15     tasklet_schedule(&xxx_tasklet);
16     ...
17 }
18
19 /*设备驱动模块加载函数*/
20 int __init xxx_init(void)
21 {
22     ...
23     /*申请中断*/
24     result = request_irq(xxx_irq, xxx_interrupt,
25         IRQF_DISABLED, "xxx", NULL);
26     ...
27     return IRQ_HANDLED;
28 }
29
30 /*设备驱动模块卸载函数*/
31 void __exit xxx_exit(void)
32 {
33     ...
34     /*释放中断*/
35     free_irq(xxx_irq, xxx_interrupt);
36     ...
37 }
```

上述程序在模块加载函数中申请中断（第 24~25 行），并在模块卸载函数中释放它（第 35 行）。对应于 xxx_irq 的中断处理程序被设置为 xxx_interrupt() 函数，在这个函数中，第 15 行的 tasklet_schedule(&xxx_tasklet) 调度被定义的 tasklet 函数 xxx_do_tasklet() 在适当的时候得到执行。

2. 工作队列

工作队列的使用方法和 tasklet 非常相似，下面的代码用于定义一个工作队列和一个底半部执行函数：

```
struct work_struct my_wq; /*定义一个工作队列*/
void my_wq_func(unsigned long); /*定义一个处理函数*/
```

通过 INIT_WORK() 可以初始化这个工作队列并将工作队列与处理函数绑定：

```
INIT_WORK(&my_wq, (void (*)(void *)) my_wq_func, NULL);
/*初始化工作队列并将其与处理函数绑定*/
```

与 tasklet_schedule() 对应的用于调度工作队列执行的函数为 schedule_work()，如：

```
schedule_work(&my_wq); /*调度工作队列执行*/
```

与代码清单 10.2 对应的使用工作队列处理中断底半部的设备驱动程序模板如代码清单 10.3 所示（仅包含与中断相关的部分）。

代码清单 10.3 工作队列使用模板

```
1 /*定义工作队列和关联函数*/
2 struct work_struct xxx_wq;
3 void xxx_do_work(unsigned long);
4
5 /*中断处理底半部*/
6 void xxx_do_work(unsigned long)
```

```

7 {
8   ...
9 }
10
11 /*中断处理顶半部*/
12 irqreturn_t xxx_interrupt(int irq, void *dev_id, struct pt_regs *regs)
13 {
14   ...
15   schedule_work(&xxx_wq);
16   ...
17   return IRQ_HANDLED;
18 }
19
20 /*设备驱动模块加载函数*/
21 int xxx_init(void)
22 {
23   ...
24   /*申请中断*/
25   result = request_irq(xxx_irq, xxx_interrupt,
26                       IRQF_DISABLED, "xxx", NULL);
27   ...
28   /*初始化工作队列*/
29   INIT_WORK(&xxx_wq, (void (*)(void *)) xxx_do_work, NULL);
30   ...
31 }
32
33 /*设备驱动模块卸载函数*/
34 void xxx_exit(void)
35 {
36   ...
37   /*释放中断*/
38   free_irq(xxx_irq, xxx_interrupt);
39   ...
40 }

```

与代码清单 10.2 不同的是，上述程序在设计驱动模块加载函数中增加了初始化工作队列的代码（第 29 行）。

尽管 Linux 社区多建议在设备第一次打开时才申请设备的中断并在最后一次关闭时释放中断以尽量减少中断被这个设备占用的时间，但是，许多情况下，驱动工程师还是将中断申请和释放的工作放在了设备驱动的模块加载和卸载函数中。

3. 软中断

软中断（softirq）也是一种传统的底半部处理机制，它的执行时机通常是顶半部返回的时候，tasklet 是基于软中断实现的，因此也运行于软中断上下文。

在 Linux 内核中，用 `softirq_action` 结构体表征一个软中断，这个结构体中包含软中断处理函数指针和传递给该函数的参数。使用 `open_softirq()` 函数可以注册软中断对应的处理函数，而 `raise_softirq()` 函数可以触发一个软中断。

软中断和 tasklet 运行于软中断上下文，仍然属于原子上下文的一种，而工作队列则运行于进程上下文。因此，软中断和 tasklet 处理函数中不能睡眠，而工作队列处理函数中允许睡眠。

`local_bh_disable()` 和 `local_bh_enable()` 是内核中用于禁止和使能软中断和 tasklet 底半部机制的函数。



内核中采用 `softirq` 的地方包括 `HI_SOFTIRQ`、`TIMER_SOFTIRQ`、`NET_TX_SOFTIRQ`、`NET_RX_SOFTIRQ`、`SCSI_SOFTIRQ`、`TASKLET_SOFTIRQ` 等, 一般来说, 驱动的编写者不会也不宜直接使用 `softirq`。

第 9 章异步通知所基于的信号也类似于中断, 现在, 总结一下硬中断、软中断和信号的区别: 硬中断是外部设备对 CPU 的中断, 软中断是中断底半部的一种处理机制, 而信号则是由内核 (或其他进程) 对某个进程的中断。在论及系统调用的场合, 人们也常说通过软中断 (例如 ARM 为 `swi`) 陷入内核, 此时软中断的概念是指由软件指令引发的中断, 和我们这个地方说的 `softirq` 是两个完全不同的概念。

10.3.4 实例: S3C6410 实时钟中断

S3C6410 处理器内部集成了实时钟 (RTC) 模块, 该模块能够在系统断电的情况下由后备电池供电继续工作, 其主要功能相对于一个时钟, 记录年、月、日、时、分、秒等。S3C6410 的 RTC 可产生两种中断: 周期节拍 (`tick`) 中断和报警 (`alarm`) 中断, 前者相当于一个周期性的定时器, 后者相当于一个“闹钟”, 它在预先设定的时间到来时产生中断。

S3C6410 实时钟设备驱动 (`drivers/rtc/rtc-s3c.c`, 为 S3C2410、S3C64XX、S5PC1XX、S5P64XX 多种 CPU 共享) 的 `open()` 函数中, 会申请它将要使用的中断, 如代码清单 10.4 所示。

代码清单 10.4 S3C6410 实时钟驱动 `open()` 函数

```
1 static int s3c_rtc_open(struct device *dev)
2 {
3     struct platform_device *pdev = to_platform_device(dev);
4     struct rtc_device *rtc_dev = platform_get_drvdata(pdev);
5     int ret;
6     /*申请 alarm 中断*/
7     ret = request_irq(s3c_rtc_alarmno, s3c_rtc_alarmirq,
8                     IRQF_DISABLED, "s3c2410-rtc alarm", rtc_dev);
9
10    if (ret) {
11        dev_err(dev, "IRQ%d error %d\n", s3c_rtc_alarmno, ret);
12        return ret;
13    }
14
15    /*申请 tick 中断*/
16    ret = request_irq(s3c_rtc_tickno, s3c_rtc_tickirq,
17                    IRQF_DISABLED, "s3c2410-rtc tick", rtc_dev);
18
19    if (ret) {
20        dev_err(dev, "IRQ%d error %d\n", s3c_rtc_tickno, ret);
21        goto tick_err;
22    }
23
24    return ret;
25
26    tick_err:
27    free_irq(s3c_rtc_alarmno, rtc_dev);
28    return ret;
29 }
```

S3C6410 实时钟设备驱动的 `release()` 函数中，会释放它将要使用的中断，如代码清单 10.5 所示。

代码清单 10.5 S3C6410 实时钟驱动 `release()` 函数

```
1 static void s3c_rtc_release(struct device *dev)
2 {
3     struct platform_device *pdev = to_platform_device(dev);
4     struct rtc_device *rtc_dev = platform_get_drvdata(pdev);
5
6     s3c_rtc_setpie(dev, 0);
7     /*释放中断*/
8     free_irq(s3c_rtc_alarmno, rtc_dev);
9     free_irq(s3c_rtc_tickno, rtc_dev);
10 }
```

S3C6410 实时钟驱动的中断处理比较简单，没有明确地分为上下两个半部，而只存在顶半部，如代码清单 10.6 所示。

代码清单 10.6 S3C6410 实时钟驱动中断处理程序

```
1 static irqreturn_t s3c_rtc_alarmirq(int irq, void *id)
2 {
3     struct rtc_device *rdev = id;
4
5     rtc_update_irq(rdev, 1, RTC_AF | RTC_IRQF);
6
7     s3c_rtc_set_bit_byte(s3c_rtc_base, S3C2410_INTP, S3C2410_INTP_ALM);
8
9     return IRQ_HANDLED;
10 }
11
12 static irqreturn_t s3c_rtc_tickirq(int irq, void *id)
13 {
14     struct rtc_device *rdev = id;
15
16     rtc_update_irq(rdev, 1, RTC_PF | RTC_IRQF);
17
18     s3c_rtc_set_bit_byte(s3c_rtc_base, S3C2410_INTP, S3C2410_INTP_TIC);
19
20     return IRQ_HANDLED;
21 }
```

上述代码中调用的 `rtc_update_irq()` 函数定义于 `drivers/rtc/interface.c` 文件中，被各种实时钟驱动共享，如代码清单 10.7 所示。

代码清单 10.7 实时钟更新 `rtc_update_irq()` 函数

```
1 void rtc_update_irq(struct rtc_device *rtc,
2                     unsigned long num, unsigned long events)
3 {
4     spin_lock(&rtc->irq_lock);
5     rtc->irq_data = (rtc->irq_data + (num << 8)) | events;
6     spin_unlock(&rtc->irq_lock);
7
8     spin_lock(&rtc->irq_task_lock);
9     if (rtc->irq_task)
10         rtc->irq_task->func(rtc->irq_task->private_data);
```



```
11     spin_unlock(&rtc->irq_task_lock);
12
13     wake_up_interruptible(&rtc->irq_queue);
14     kill_fasync(&rtc->async_queue, SIGIO, POLL_IN);
15 }
```

上述中断处理程序并没有底半部（或者说没有严格意义上的 tasklet、工作队列或软中断底半部），实际上，它只是唤醒一个等待队列 `rtc->irq_queue` 并发出一个 SIGIO 信号，而这个等待队列的唤醒也将导致一个阻塞的进程被执行（这个阻塞的进程可看作底半部）。现在我们看到，等待队列可以作为中断处理程序顶半部和进程同步的一种良好机制。但是，任何情况下，都不能在顶半部等待一个等待队列，而只能唤醒。

10.4 中断共享

多个设备共享一根硬件中断线的情况在实际的硬件系统中广泛存在，Linux 2.6 支持这种中断共享。下面是中断共享的使用方法。

(1) 共享中断的多个设备在申请中断时，都应该使用 `IRQF_SHARED` 标志，而且一个设备以 `IRQF_SHARED` 申请某中断成功的前提是该中断未被申请，或该中断虽然被申请了，但是之前申请该中断的所有设备也都以 `IRQF_SHARED` 标志申请该中断。

(2) 尽管内核模块可访问的全局地址都可以作为 `request_irq(..., void *dev_id)` 的最后一个参数 `dev_id`，但是设备结构体指针显然是可传入的最佳参数。

(3) 在中断到来时，会遍历执行共享此中断的所有中断处理程序，直到某一个函数返回 `IRQ_HANDLED`。在中断处理程序顶半部中，应迅速地根据硬件寄存器中的信息比照传入的 `dev_id` 参数判断是否是本设备的中断，若不是，应迅速返回 `IRQ_NONE`，如图 10.3 所示。

代码清单 10.8 给出了使用共享中断的设备驱动程序的模板（仅包含与共享中断机制相关的部分）。

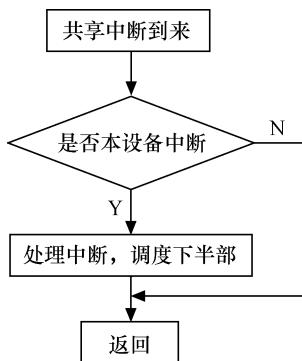


图 10.3 共享中断的处理

代码清单 10.8 共享中断编程模板

```
1  /*中断处理顶半部*/
2  irqreturn_t xxx_interrupt(int irq, void *dev_id, struct pt_regs *regs)
3  {
4      ...
5      int status = read_int_status(); /*获知中断源*/
6      if(!is_myint(dev_id, status)) /*判断是否是本设备中断*/
7          return IRQ_NONE; /*不是本设备中断，立即返回*/
8
9      /* 是本设备中断，进行处理 */
10     ...
11     return IRQ_HANDLED; /* 返回 IRQ_HANDLED 表明中断已被处理 */
12 }
13
```

```

14 /*设备驱动模块加载函数*/
15 int xxx_init(void)
16 {
17     ...
18     /*申请共享中断*/
19     result = request_irq(sh_irq, xxx_interrupt,
20         IRQF_SHARED, "xxx", xxx_dev);
21     ...
22 }
23
24 /*设备驱动模块卸载函数*/
25 void xxx_exit(void)
26 {
27     ...
28     /*释放中断*/
29     free_irq(xxx_irq, xxx_interrupt);
30     ...
31 }

```

10.5 内核定时器

10.5.1 内核定时器编程

软件意义上的定时器最终依赖硬件定时器来实现，内核在时钟中断发生后执行检测各定时器是否到期，到期后的定时器处理函数将作为软中断在底半部执行。实质上，时钟中断处理程序会唤起 `TIMER_SOFTIRQ` 软中断，运行当前处理器上到期的所有定时器。

在 Linux 设备驱动编程中，可以利用 Linux 内核中提供的一组函数和数据结构来完成定时触发工作或者完成某周期性的事务。这组函数和数据结构使得驱动工程师多数情况下不用关心具体的软件定时器究竟对应着怎样的内核和硬件行为。

Linux 内核所提供的用于操作定时器的数据结构和函数如下。

1. timer_list

在 Linux 内核中，`timer_list` 结构体的一个实例对应一个定时器，如代码清单 10.9 所示。

代码清单 10.9 timer_list 结构体

```

1 struct timer_list {
2     struct list_head entry; /* 定时器列表 */
3     unsigned long expires; /*定时器到期时间*/
4     void (*function)(unsigned long); /* 定时器处理函数 */
5     unsigned long data; /* 作为参数被传入定时器处理函数 */
6     struct timer_base_s *base;
7     ...
8 };

```

当定时器期满后，其中第 5 行的 `function()` 成员将被执行，而第 4 行的 `data` 成员则是传入其中的参数，第 3 行的 `expires` 则是定时器到期的时间（jiffies）。

如下代码定义一个名为 `my_timer` 的定时器：

```
struct timer_list my_timer;
```



2. 初始化定时器

```
void init_timer(struct timer_list * timer);
```

上述 `init_timer()` 函数初始化 `timer_list` 的 `entry` 的 `next` 为 `NULL`，并给 `base` 指针赋值。

`TIMER_INITIALIZER(_function, _expires, _data)` 宏用于赋值定时器结构体的 `function`、`expires`、`data` 和 `base` 成员，这个宏的定义为：

```
#define TIMER_INITIALIZER(_function, _expires, _data) { \
    .entry = { .prev = TIMER_ENTRY_STATIC }, \
    .function = (_function), \
    .expires = (_expires), \
    .data = (_data), \
    .base = &boot_tvec_bases, \
}
```

`DEFINE_TIMER(_name, _function, _expires, _data)` 宏是定义并初始化定时器成员的“快捷方式”，这个宏定义为：

```
#define DEFINE_TIMER(_name, _function, _expires, _data) \
    struct timer_list _name = \
        TIMER_INITIALIZER(_function, _expires, _data)
```

此外，`setup_timer()` 也可用于初始化定时器并赋值其成员，其源代码为：

```
static inline void setup_timer(struct timer_list * timer, \
    void (*function)(unsigned long), \
    unsigned long data) \
{ \
    timer->function = function; \
    timer->data = data; \
    init_timer(timer); \
}
```

3. 增加定时器

```
void add_timer(struct timer_list * timer);
```

上述函数用于注册内核定时器，将定时器加入到内核动态定时器链表中。

4. 删除定时器

```
int del_timer(struct timer_list * timer);
```

上述函数用于删除定时器。

`del_timer_sync()` 是 `del_timer()` 的同步版，在删除一个定时器时需等待其被处理完，因此该函数的调用不能发生在中断上下文。

5. 修改定时器的 expire

```
int mod_timer(struct timer_list *timer, unsigned long expires);
```

上述函数用于修改定时器的到期时间，在新的被传入的 `expires` 到来后才会执行定时器函数。

代码清单 10.10 给出了一个完整的内核定时器使用模板，大多数情况下，设备驱动都如这个模板那样使用定时器。

代码清单 10.10 内核定时器使用模板

```
1 /*xxx 设备结构体*/ \
2 struct xxx_dev { \
3     struct cdev cdev; \
4     ... \
5     timer_list xxx_timer; /*设备要使用的定时器*/ \
6 };
```



```

7
8 /*xxx 驱动中的某函数*/
9 xxx_func1 (...)
10 {
11     struct xxx_dev *dev = filp->private_data;
12     ...
13     /*初始化定时器*/
14     init_timer(&dev->xxx_timer);
15     dev->xxx_timer.function = &xxx_do_timer;
16     dev->xxx_timer.data = (unsigned long)dev;
17                     /*设备结构体指针作为定时器处理函数参数*/
18     dev->xxx_timer.expires = jiffies + delay;
19     /*添加（注册）定时器*/
20     add_timer(&dev->xxx_timer);
21     ...
22 }
23
24 /*xxx 驱动中的某函数*/
25 xxx_func2 (...)
26 {
27     ...
28     /*删除定时器*/
29     del_timer (&dev->xxx_timer);
30     ...
31 }
32
33 /*定时器处理函数*/
34 static void xxx_do_timer(unsigned long arg)
35 {
36     struct xxx_device *dev = (struct xxx_device *) (arg);
37     ...
38     /*调度定时器再执行*/
39     dev->xxx_timer.expires = jiffies + delay;
40     add_timer(&dev->xxx_timer);
41     ...
42 }

```

从代码清单第 18、39 行可以看出，定时器的到期时间往往是在目前 `jiffies` 的基础是添加一个时延，若为 Hz，则表示延迟 1s。

在定时器处理函数中，在做完相应的工作后，往往会延后 `expires` 并将定时器再次添加到内核定时器链表，以便定时器能再次被触发。

10.5.2 内核中延迟的工作 `delayed_work`

注意，对于这种周期性的任务，Linux 内核还提供了一套封装好的快捷机制，其本质利用工作队列和定时器实现，这套快捷机制就是 `delayed_work`，`delayed_work` 结构体的定义如代码清单 10.11 所示。

代码清单 10.11 `delayed_work` 结构体

```

1 struct delayed_work {
2     struct work_struct work;
3     struct timer_list timer;
4 };
5 struct work_struct {

```



```
6         atomic_long_t data;
7 #define WORK_STRUCT_PENDING 0
8 #define WORK_STRUCT_FLAG_MASK (3UL)
9 #define WORK_STRUCT_WQ_DATA_MASK (~WORK_STRUCT_FLAG_MASK)
10        struct list_head entry;
11        work_func_t func;
12 #ifdef CONFIG_LOCKDEP
13        struct lockdep_map lockdep_map;
14 #endif
15 };
```

我们可以通过如下函数调度一个 `delayed_work` 在指定的延时后执行:

```
int schedule_delayed_work(struct delayed_work *work, unsigned long delay);
```

当指定的 `delay` 到来时 `delayed_work` 结构体中 `work` 成员的 `work_func_t` 类型成员 `func()` 会被执行。`work_func_t` 类型定义为:

```
typedef void (*work_func_t)(struct work_struct *work);
```

其中 `delay` 参数的单位是 `jiffies`, 因此一种常见的用法如下:

```
schedule_delayed_work(&work, msecs_to_jiffies(poll_interval));
```

其中的 `msecs_to_jiffies()` 用于将毫秒转化为 `jiffies`。

如果要周期性的执行任务, 通常会在 `delayed_work` 的工作函数中再次调用 `schedule_delayed_work()`, 周而复始。

如下函数用来取消 `delayed_work`:

```
int cancel_delayed_work(struct delayed_work *work);
int cancel_delayed_work_sync(struct delayed_work *work);
```

10.5.3 实例: 秒字符设备

下面我们编写一个字符设备“second”(即“秒”)的驱动, 它在被打开的时候初始化一个定时器并将其添加到内核定时器链表, 每秒输出一当前时的 `jiffies` (为此, 定时器处理函数中每次都要修改新的 `expires`), 整个程序如代码清单 10.11 所示。

代码清单 10.12 使用内核定时器的 second 字符设备驱动

```
1 #include <linux/module.h>
2 #include <linux/types.h>
3 #include <linux/fs.h>
4 #include <linux/errno.h>
5 #include <linux/mm.h>
6 #include <linux/sched.h>
7 #include <linux/init.h>
8 #include <linux/cdev.h>
9 #include <asm/io.h>
10 #include <asm/system.h>
11 #include <asm/uaccess.h>
12
13 #define SECOND_MAJOR 248 /*预设的 second 的主设备号*/
14
15 static int second_major = SECOND_MAJOR;
16
17 /*second 设备结构体*/
18 struct second_dev {
19     struct cdev cdev; /*cdev 结构体*/
```

```

20     atomic_t counter; /* 一共经历了多少秒? */
21     struct timer_list s_timer; /*设备要使用的定时器*/
22 };
23
24 struct second_dev *second_devp; /*设备结构体指针*/
25
26 /*定时器处理函数*/
27 static void second_timer_handle(unsigned long arg)
28 {
29     mod_timer(&second_devp->s_timer, jiffies + Hz);
30     atomic_inc(&second_devp->counter);
31
32     printk(KERN_NOTICE "current jiffies is %ld\n", jiffies);
33 }
34
35 /*文件打开函数*/
36 int second_open(struct inode *inode, struct file *filp)
37 {
38     /*初始化定时器*/
39     init_timer(&second_devp->s_timer);
40     second_devp->s_timer.function = &second_timer_handle;
41     second_devp->s_timer.expires = jiffies + Hz;
42
43     add_timer(&second_devp->s_timer); /*添加(注册)定时器*/
44
45     atomic_set(&second_devp->counter, 0); //计数清0
46
47     return 0;
48 }
49 /*文件释放函数*/
50 int second_release(struct inode *inode, struct file *filp)
51 {
52     del_timer(&second_devp->s_timer);
53
54     return 0;
55 }
56
57 /*读函数*/
58 static ssize_t second_read(struct file *filp, char __user *buf, size_t count,
59     loff_t *ppos)
60 {
61     int counter;
62
63     counter = atomic_read(&second_devp->counter);
64     if(put_user(counter, (int*)buf))
65         return -EFAULT;
66     else
67         return sizeof(unsigned int);
68 }
69
70 /*文件操作结构体*/
71 static const struct file_operations second_fops = {
72     .owner = THIS_MODULE,
73     .open = second_open,
74     .release = second_release,

```



```
75     .read = second_read,
76 };
77
78 /*初始化并注册 cdev*/
79 static void second_setup_cdev(struct second_dev *dev, int index)
80 {
81     int err, devno = MKDEV(second_major, index);
82
83     cdev_init(&dev->cdev, &second_fops);
84     dev->cdev.owner = THIS_MODULE;
85     err = cdev_add(&dev->cdev, devno, 1);
86     if (err)
87         printk(KERN_NOTICE "Error %d adding LED%d", err, index);
88 }
89
90 /*设备驱动模块加载函数*/
91 int second_init(void)
92 {
93     int ret;
94     dev_t devno = MKDEV(second_major, 0);
95
96     /* 申请设备号*/
97     if (second_major)
98         ret = register_chrdev_region(devno, 1, "second");
99     else { /* 动态申请设备号 */
100         ret = alloc_chrdev_region(&devno, 0, 1, "second");
101         second_major = MAJOR(devno);
102     }
103     if (ret < 0)
104         return ret;
105     /* 动态申请设备结构体的内存*/
106     second_devp = kmalloc(sizeof(struct second_dev), GFP_KERNEL);
107     if (!second_devp) { /*申请失败*/
108         ret = - ENOMEM;
109         goto fail_malloc;
110     }
111
112     memset(second_devp, 0, sizeof(struct second_dev));
113
114     second_setup_cdev(second_devp, 0);
115
116     return 0;
117
118 fail_malloc:
119     unregister_chrdev_region(devno, 1);
120     return ret;
121 }
122
123 /*模块卸载函数*/
124 void second_exit(void)
125 {
126     cdev_del(&second_devp->cdev); /*注销 cdev*/
127     kfree(second_devp); /*释放设备结构体内存*/
128     unregister_chrdev_region(MKDEV(second_major, 0), 1); /*释放设备号*/
129 }
```

```

130
131 MODULE_AUTHOR("Barry Song <21cnbao@gmail.com>");
132 MODULE_LICENSE("Dual BSD/GPL");
133
134 module_param(second_major, int, S_IRUGO);
135
136 module_init(second_init);
137 module_exit(second_exit);

```

在 `second` 的 `open()` 函数中，将启动定时器，此后每 1s 会再次运行定时器处理函数，在 `second` 的 `release()` 函数中，定时器被删除。

`second_dev` 结构体中的原子变量 `counter` 用于秒计数，每次在定时器处理函数中将被 `atomic_inc()` 调用原子的增 1，`second` 的 `read()` 函数会将这个值返回给用户空间。

`/home/lihacker/develop/svn/ldd6410-read-only/training/kernel/drivers/second` 包含了 `second` 设备驱动以及 `second_test.c` 用户空间测试程序，运行 `make` 命令编译得到 `second.ko` 和 `second_test`，加载 `second.ko` 内核模块并创建 “`/dev/second`” 设备文件结点：

```

root@lihacker-laptop:/home/lihacker/develop/svn/ldd6410-read-only/training/kernel/d
rivers/second# mknod /dev/second c 248 0

```

代码清单 10.13 给出了 `second_test.c` 这个应用程序，它打开 “`/dev/second`”，其后不断地读取自打开 “`/dev/second`” 设备文件以来经历的秒数。

代码清单 10.13 `second` 设备用户空间测试程序

```

1 #include ...
2
3 main()
4 {
5     int fd;
6     int counter = 0;
7     int old_counter = 0;
8
9     /*打开/dev/second 设备文件*/
10    fd = open("/dev/second", O_RDONLY);
11    if (fd != - 1) {
12        while (1) {
13            read(fd,&counter, sizeof(unsigned int));/* 读目前经历的秒数 */
14            if(counter!=old_counter) {
15                printf("seconds after open /dev/second :%d\n",counter);
16                old_counter = counter;
17            }
18        }
19    } else {
20        printf("Device open failure\n");
21    }
22 }

```

运行 `second_test` 后，内核将不断地输出目前的 `jiffies` 值：

```

[44335.554313] current jiffies is 11008888
[44336.553166] current jiffies is 11009138
[44337.553175] current jiffies is 11009388
[44338.552383] current jiffies is 11009638
[44339.552321] current jiffies is 11009888
...

```



而应用程序将不断输出自打开/dev/second 以来经历的秒数:

```
root@lihacker-laptop:/home/lihacker/develop/svn/ldd6410-read-only/training/kernel/d
rivers/second# ./second_test
seconds after open /dev/second :1
seconds after open /dev/second :2
seconds after open /dev/second :3
seconds after open /dev/second :4
seconds after open /dev/second :5
...
```

10.6 内核延时

10.6.1 短延迟

Linux 内核中提供了如下 3 个函数分别进行纳秒、微秒和毫秒延迟:

```
void ndelay(unsigned long nsecs);
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```

上述延迟的实现原理本质上是忙等待,它根据 CPU 频率进行一定次数的循环。有时候,人们在软件中进行这样的延迟:

```
void delay(unsigned int time)
{
    while (time--);
}
```

ndelay()、udelay()和 mdelay()函数的实现方式机理与此类似。内核在启动时,会运行一个延迟测试程序(delay loop calibration),计算出 lpj (loops per jiffy),例如对于 LDD6410 电路板而言,内核启动时会打印:

```
Calibrating delay loop... 530.84 BogoMIPS (lpj=1327104)
```

毫秒时延(以及更大的秒时延)已经比较大了,在内核中,最好不要直接使用 mdelay()函数,这将无谓地耗费 CPU 资源,对于毫秒级以上时延,内核提供了下述函数:

```
void msleep(unsigned int millisecs);
unsigned long msleep_interruptible(unsigned int millisecs);
void ssleep(unsigned int seconds);
```

上述函数将使得调用它的进程睡眠参数指定的时间,msleep()、ssleep()不能被打断,而 msleep_interruptible()则可以被打断。



受系统 Hz 以及进程调度的影响,msleep()类似函数的精度是有限的。

10.6.2 长延迟

内核中进行延迟的一个很直观的方法是比较当前的 jiffies 和目标 jiffies (设置为当前 jiffies 加

上时间间隔的 jiffies)，直到未来的 jiffies 达到目标 jiffies。代码清单 10.14 给出了使用忙等待先延迟 100 个 jiffies 再延迟 2s 的实例。

代码清单 10.14 忙等待时延实例

```
1 /*延迟 100 个 jiffies*/
2 unsigned long delay = jiffies + 100;
3 while (time_before(jiffies, delay));
4
5 /*再延迟 2s*/
6 unsigned long delay = jiffies + 2*Hz;
7 while (time_before(jiffies, delay));
```

与 `time_before()` 对应的还有一个 `time_after()`，它们在内核中定义为（实际上只是将传入的未来时间 jiffies 和被调用时的 jiffies 进行一个简单的比较）：

```
#define time_after(a,b) \
    (typecheck(unsigned long, a) && \
     typecheck(unsigned long, b) && \
     ((long)(b) - (long)(a) < 0))
#define time_before(a,b) time_after(b,a)
```

为了防止 `time_before()` 和 `time_after()` 的比较过程中编译器对 jiffies 的优化，内核将其定义为 volatile 变量，这将保证它每次都被重新读取。

10.6.3 睡着延迟

睡着延迟无疑是比忙等待更好的方式，睡着延迟在等待的时间到来之间进程处于睡眠状态，CPU 资源被其他进程使用。`schedule_timeout()` 可以使当前任务睡眠指定的 jiffies 之后重新被调度执行，`msleep()` 和 `msleep_interruptible()` 在本质上都是依靠包含了 `schedule_timeout()` 的 `schedule_timeout_uninterruptible()` 和 `schedule_timeout_interruptible()` 实现的，如代码清单 10.15 所示。

代码清单 10.15 `schedule_timeout()` 的使用

```
1 void msleep(unsigned int msecs)
2 {
3     unsigned long timeout = msecs_to_jiffies(msecs) + 1;
4
5     while (timeout)
6         timeout = schedule_timeout_uninterruptible(timeout);
7 }
8
9 unsigned long msleep_interruptible(unsigned int msecs)
10 {
11     unsigned long timeout = msecs_to_jiffies(msecs) + 1;
12
13     while (timeout && !signal_pending(current))
14         timeout = schedule_timeout_interruptible(timeout);
15     return jiffies_to_msecs(timeout);
16 }
```

实际上，`schedule_timeout()` 的实现原理是向系统添加一个定时器，在定时器处理函数中唤醒参数对应的进程。

代码清单 10.15 第 6 行和第 14 行分别调用 `schedule_timeout_uninterruptible()` 和 `schedule_timeout_`



interruptible(), 这两个函数的区别在于前者在调用 schedule_timeout()之前置进程状态为 TASK_INTERRUPTIBLE, 后者置进程状态为 TASK_UNINTERRUPTIBLE, 如代码清单 10.16 所示。

代码清单 10.16 schedule_timeout_interruptible()和 schedule_timeout_uninterruptible()

```
1 signed long __sched schedule_timeout_interruptible(signed long timeout)
2 {
3     __set_current_state(TASK_INTERRUPTIBLE);
4     return schedule_timeout(timeout);
5 }
6
7 signed long __sched schedule_timeout_uninterruptible(signed long timeout)
8 {
9     __set_current_state(TASK_UNINTERRUPTIBLE);
10    return schedule_timeout(timeout);
11 }
```

另外, 下面两个函数可以将当前进程添加到等待队列中, 从而在等待队列上睡眠。当超时发生时, 进程将被唤醒 (后者可以在超时前被打断):

```
sleep_on_timeout(wait_queue_head_t *q, unsigned long timeout);
interruptible_sleep_on_timeout(wait_queue_head_t*q, unsigned long timeout);
```

10.7 总结

Linux 的中断处理分为两个半部, 顶半部处理紧急的硬件操作, 底半部处理不紧急的耗时操作。tasklet 和工作队列都是调度中断底半部的良好机制, tasklet 基于软中断实现。内核定时器也依靠软中断实现。

内核中的延时可以采用忙等待或睡眠等待, 为了充分利用 CPU 资源, 使系统有更好的吞吐性能, 在对延迟时间的要求并不是很精确的情况下, 睡眠等待通常是值得推荐的。而 ndelay()、udelay() 忙等待机制在驱动中通常是为了配合硬件上短时延迟要求。

LINUX

第11章

内存与 I/O 访问

由于 Linux 系统中提供了复杂的内存管理功能，所以内存的概念在 Linux 系统中变得相对复杂，出现了常规内存、高端内存、虚拟地址、逻辑地址、总线地址、物理地址、I/O 内存、设备内存、预留内存等概念。本章将系统地讲解内存和 I/O 的访问编程，带您走出内存和 I/O 的概念迷宫。

11.1 节讲解内存和 I/O 的硬件机制，主要涉及内存空间、I/O 空间和 MMU。

11.2 节讲解 Linux 的内存管理、内存区域的分布、常规内存与高端内存的区别。

11.3 节讲解 Linux 内存存取的方法，主要涉及内存动态申请以及通过虚拟地址存取物理地址的方法。

11.4 节讲解设备 I/O 内存和 I/O 端口的访问流程，这一节对于编写设备驱动意义重大，设备驱动使用此节的方法访问物理设备。

11.5 节讲解设备驱动中的 DMA 与 CACHE 一致性问题以及 DMA 编程方法。





11.1 CPU 与内存和 I/O

11.1.1 内存空间与 I/O 空间

在 X86 处理器中存在着 I/O 空间的概念, I/O 空间是相对于内存空间而言的, 它通过特定的指令 in、out 来访问。端口号标识了外设的寄存器地址。Intel 语法的 in、out 指令格式如下:

```
IN 累加器, {端口号|DX}
OUT {端口号|DX}, 累加器
```

目前, 大多数嵌入式微控制器如 ARM、PowerPC 等中并不提供 I/O 空间, 而仅存在内存空间。内存空间可以直接通过地址、指针来访问, 程序和程序运行中使用的变量和其他数据都存在于内存空间中。

内存地址可以直接由 C 语言指针操作, 例如在 186 处理器中执行如下代码:

```
unsigned char *p = (unsigned char *)0xF000FF00;
*p=11;
```

以上程序的意义为在绝对地址 0xF0000+0xFF00 (186 处理器使用 16 位段地址和 16 位偏移地址) 写入 11。

而在 ARM、PowerPC 等未采用段地址的处理器中, p 指向的内存空间就是 0xF000FF00, 而 *p = 11 就是在该地址写入 11。

再如, 186 处理器启动后会在绝对地址 0xFFFF0 (对应 C 语言指针是 0xF000FFF0, 0xF000 为段地址, 0xFFFF0 为段内偏移) 执行, 请看下面的代码:

```
typedef void (*lpFunction) ( ); /* 定义一个无参数、无返回类型的函数指针类型*/
lpFunction lpReset = (lpFunction)0xF000FFF0; /* 定义一个函数指针, 指向*/
/* CPU 启动后所执行第一条指令的位置 */
lpReset(); /* 调用函数 */
```

在以上程序中, 没有定义任何一个函数实体, 但是程序中却执行了这样的函数调用: lpReset(), 它实际上起到了“软重启”的作用, 跳转到 CPU 启动后第一条要执行的指令的位置。因此, 可以通过函数指针调用一个没有函数体的“函数”, 本质上只是换一个地址开始执行。

即便是在 X86 处理器中, 虽然提供了 I/O 空间, 如果由我们自己设计电路板, 外设仍然可以只挂接在内存空间。此时, CPU 可以像访问一个内存单元那样访问外设 I/O 端口, 而不需要设立专门的 I/O 指令。因此, 内存空间是必须的, 而 I/O 空间是可选的。图 11.1 给出了内存空间和 I/O 空间的对比。

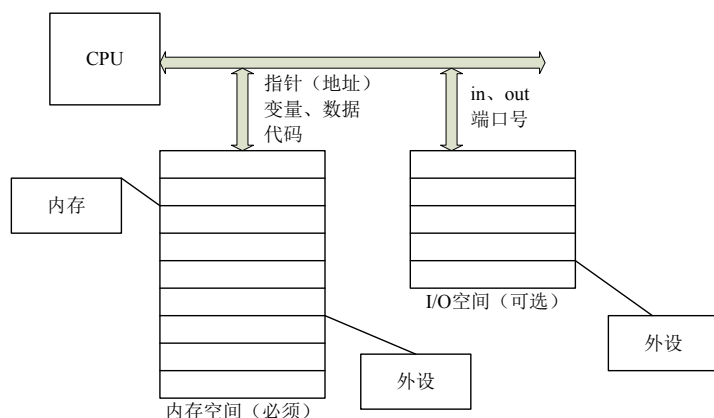


图 11.1 内存空间和 I/O 空间

11.1.2 内存管理单元 MMU

高性能处理器一般会提供一个内存管理单元（MMU），该单元辅助操作系统进行内存管理，提供虚拟地址和物理地址的映射、内存访问权限保护和 Cache 缓存控制等硬件支持。操作系统内核借助 MMU，可以让用户感觉到好像程序可以使用非常大的内存空间，从而使得编程人员在写程序时不用考虑计算机中的物理内存的实际容量。

为了理解基本的 MMU 操作原理，需先明晰几个概念。

（1）TLB：Translation Lookaside Buffer，即转换旁路缓存，TLB 是 MMU 的核心部件，它缓存少量的虚拟地址与物理地址的转换关系，是转换表的 Cache，因此也经常被称为“快表”。

（2）TTW：Translation Table walk，即转换表漫游，当 TLB 中没有缓冲对应的地址转换关系时，需要通过对内存中转换表（大多数处理器的转换表为多级页表，如图 11.2 所示）的访问来获得虚拟地址和物理地址的对应关系。TTW 成功后，结果应写入 TLB。

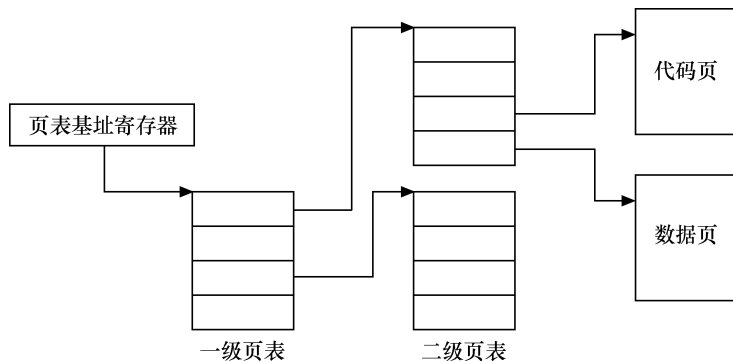


图 11.2 内存中的转换表

图 11.3 给出了一个典型的 ARM 处理器访问内存的过程，其他处理器也执行类似过程。当 ARM 要访问存储器时，MMU 先查找 TLB 中的虚拟地址表。如果 ARM 的结构支持分开的数据 TLB（DTLB）和指令 TLB（ITLB），则除取指令使用 ITLB 外，其他的都使用 DTLB。ARM 处理器的 MMU 如图 11.3 所示。

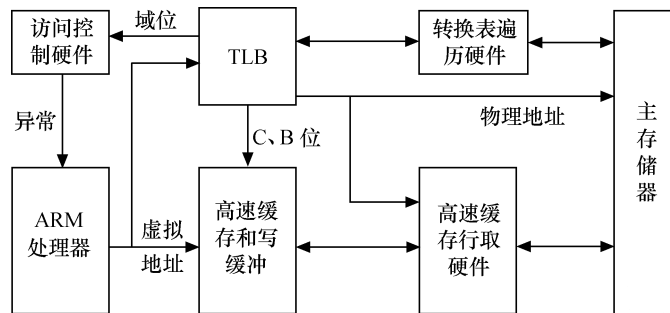


图 11.3 ARM 的内存管理单元

若 TLB 中没有虚拟地址的入口，则转换表遍历硬件从存放于主存储器中的转换表中获取地址转换信息和访问权限（即执行 TTW），同时将这些信息放入 TLB，它或者被放在一个没有使用的入口或者替换一个已经存在的入口。之后，在 TLB 条目中控制信息的控制下，当访问权限允许时，对真实物理地址的访问将在 Cache 或者在内存中发生，如图 11.4 所示。

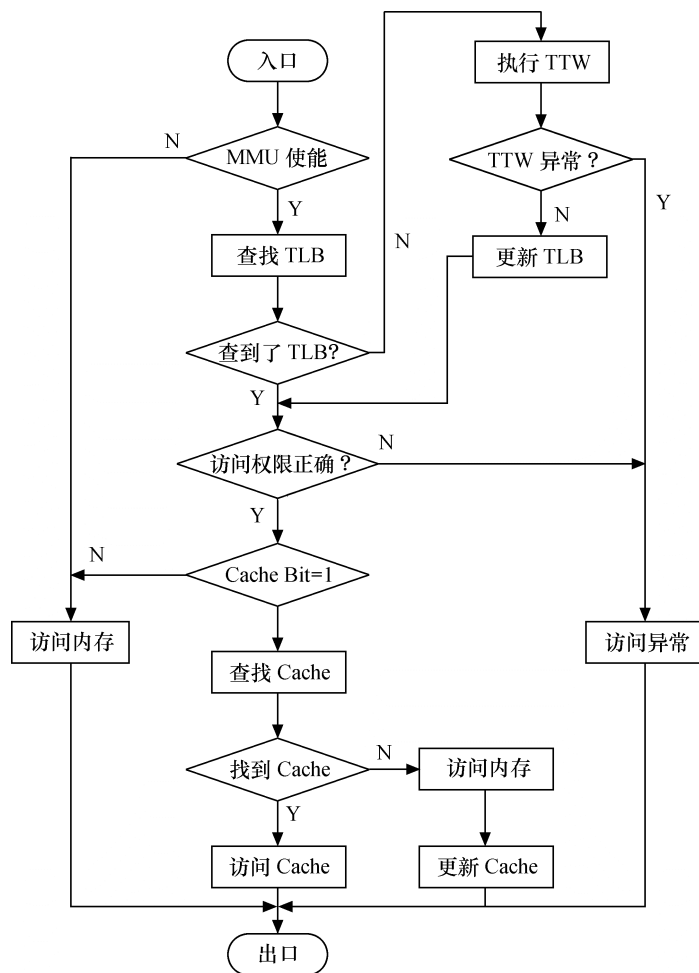


图 11.4 ARM CPU 进行数据访问的流程

ARM 中的 TLB 条目中的控制信息用于控制对对应地址的访问权限以及 Cache 的操作。

- C（高速缓存）和 B（缓冲）位被用来控制对应地址的高速缓存和写缓冲，并决定是否高速缓存。
- 访问权限和域位用来控制读写访问是否被允许。如果不允许，则 MMU 将向 ARM 处理器发送一个存储器异常，否则访问将被允许进行。

上述描述的 MMU 机制针对的虽然是 ARM 处理器，但 PowerPC、MIPS 等其他处理器也均有类似的操作。

MMU 具有虚拟地址和物理地址转换、内存访问权限保护等功能，这将使得 Linux 操作系统能单独为系统的每个用户进程分配独立的内存空间并保证用户空间不能访问内核空间的地址，为操作系统的虚拟内存管理模块提供硬件基础。

Linux 内核使用了三级页表 PGD、PMD 和 PTE，对于许多体系结构而言，PMD 这一级实际上只有一个入口。代码清单 11.1 给出了一个典型的从虚拟地址得到 PTE 的页表查询（page table walk）过程。

代码清单 11.1 Linux 的三级页表与页表查询

```
1 static int walk_page_tables(struct mm_struct *mm,
2                             unsigned long address,
3                             pte_t *pte_ret)
4 {
5     pgd_t *pgd;
6     pmd_t *pmd;
7     pte_t *ptep;
8 #ifdef HAVE_PUD_T
9     pud_t *pud;
10 #endif
11
12     pgd = pgd_offset(mm, address);
13     if (pgd_none(*pgd) || unlikely(pgd_bad(*pgd)))
14         goto out;
15
16 #ifdef HAVE_PUD_T
17     pud = pud_offset(pgd, address);
18     if (pud_none(*pud) || unlikely(pud_bad(*pud)))
19         goto out;
20
21     pmd = pmd_offset(pud, address);
22 #else
23     pmd = pmd_offset(pgd, address);
24 #endif
25     if (pmd_none(*pmd) || unlikely(pmd_bad(*pmd)))
26         goto out;
27     ptep = pte_offset_map(pmd, address);
28     if (!ptep)
29         goto out;
30
31     *pte_ret = *ptep;
32     pte_unmap(ptep);
```



```
33
34 return 0;
35 out:
36 return -1;
37 }
```

第 1 行的类型为 `struct mm_struct` 的参数 `mm` 用于描述 Linux 进程所占有的内存资源。上述代码中的 `pgd_offset`、`pmd_offset` 分别用于得到一级页表和二级页表的入口，最后通过 `pte_offset_map` 得到目标页表项。

但是，MMU 并非对所有处理器都是必须的，例如常用的 SAMSUNG 基于 ARM7TDMI 系列的 S3C44B0X 不附带 MMU，新版的 Linux 2.6 支持不带 MMU 的处理器。在嵌入式系统中，仍存在大量无 MMU 的处理器，Linux 2.6 为了更广泛地应用于嵌入式系统，融合了 `μClinux`，以支持这些 MMU-less 系统，如 Dragonball、ColdFire、Hitachi H8/300、Blackfin 等。

11.2 Linux 内存管理

对于包含 MMU 的处理器而言，Linux 系统提供了复杂的存储管理系统，使得进程所能访问的内存达到 4GB。

在 Linux 系统中，进程的 4GB 内存空间被分为两个部分——用户空间与内核空间。用户空间地址一般分布为 0~3GB（即 `PAGE_OFFSET`，在 0x86 中它等于 `0xC0000000`），这样，剩下的 3~4GB 为内核空间，如图 11.5 所示。用户进程通常情况下只能访问用户空间的虚拟地址，不能访问内核空间虚拟地址。用户进程只有通过系统调用（代表用户进程在内核态执行）等方式才可以访问到内核空间。

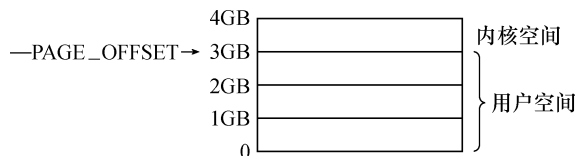


图 11.5 用户空间与内核空间

每个进程的用户空间都是完全独立、互不相干的，用户进程各自有不同的页表。而内核空间是由内核负责映射，它并不会跟着进程改变，是固定的。内核空间地址有自己对应的页表，内核的虚拟空间独立于其他程序。

Linux 中 1GB 的内核地址空间又被划分为物理内存映射区、虚拟内存分配区、高端页面映射区、专用页面映射区和系统保留映射区这几个区域，如图 11.6 所示。

一般情况下，物理内存映射区最大长度为 896MB，系统的物理内存被顺序映射在内核空间的这个区域中。当系统物理内存大于 896MB 时，超过物理内存映射区的那部分内存称为高端内存（而未超过物理内存映射区的内存通常被称为常规内存），内核在存取高端内存时必须将它们映射到高端页面映射区。

Linux 保留内核空间最顶部 `FIXADDR_TOP`~4GB 的区域作为保留区。

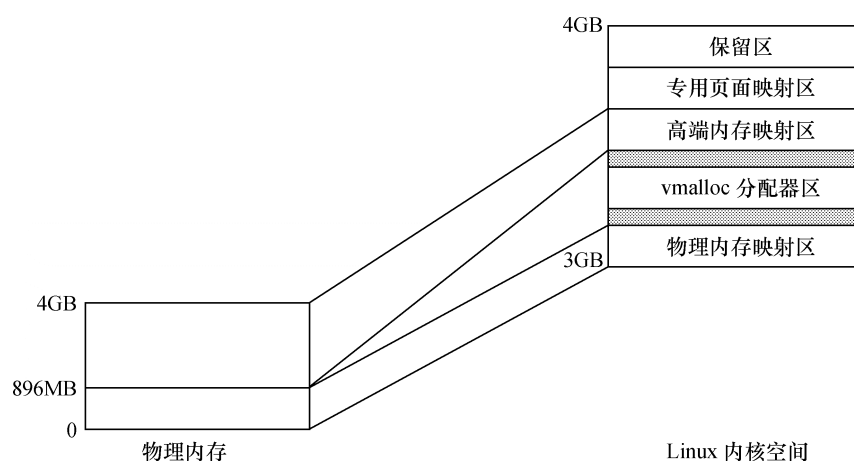


图 11.6 Linux 内核地址空间

紧接着最顶端的保留区以下的一段区域为专用页面映射区 (FIXADDR_START~FIXADDR_TOP)，它的总尺寸和每一页的用途由 `fixed_address` 枚举结构在编译时预定义，用 `__fix_to_virt(index)` 可获取专用区内预定义页面的逻辑地址。其开始地址和结束地址宏定义如下：

```
#define FIXADDR_START      (FIXADDR_TOP - __FIXADDR_SIZE)
#define FIXADDR_TOP        ((unsigned long) __FIXADDR_TOP)
#define __FIXADDR_TOP      0xfffff000
```

接下来，如果系统配置了高端内存，则位于专用页面映射区之下的就是一段高端内存映射区，其起始地址为 `PKMAP_BASE`，定义如下：

```
#define PKMAP_BASE ( (FIXADDR_BOOT_START - PAGE_SIZE*(LAST_PKMAP + 1)) & PMD_MASK )
```

其中所涉及的宏定义如下：

```
#define FIXADDR_BOOT_START (FIXADDR_TOP - __FIXADDR_BOOT_SIZE)
#define LAST_PKMAP        PTRS_PER_PTE
#define PTRS_PER_PTE      512
#define PMD_MASK          (~(PMD_SIZE-1))
# define PMD_SIZE (1UL << PMD_SHIFT)
#define PMD_SHIFT 21
```

在物理区和高端映射区之间为虚存内存分配区 (VMALLOC_START~VMALLOC_END)，用于 `vmalloc()` 函数，它的前部与物理内存映射区有一个隔离带，后部与高端映射区也有一个隔离带，`vmalloc` 区域定义如下：

```
#define VMALLOC_OFFSET (8*1024*1024)
#define VMALLOC_START (((unsigned long) high_memory +
    vmalloc_earlyreserve + 2*VMALLOC_OFFSET-1) & ~(VMALLOC_OFFSET-1))

#ifdef CONFIG_HIGHMEM /*支持高端内存*/
# define VMALLOC_END (PKMAP_BASE-2*PAGE_SIZE)
#else /*不支持高端内存*/
# define VMALLOC_END (FIXADDR_START-2*PAGE_SIZE)
#endif
```

当系统物理内存超过 4GB 时，必须使用 CPU 的扩展分页 (PAE) 模式所提供的 64 位页目录项才能存取到 4GB 以上的物理内存，这需要 CPU 的支持。加入了 PAE 功能的 Intel Pentium Pro 及其后的 CPU 允许内存最大可配置到 64GB，具备 36 位物理地址空间寻址能力。



由此可见, 在 3~4GB 之间的内核空间中, 从低地址到高地址依次为: 物理内存映射区—隔离带—vmalloc 虚拟内存分配器—隔离带—高端内存映射区—专用页面映射区—保留区。

11.3 内存存取

11.3.1 用户空间内存动态申请

在用户空间动态申请内存的函数为 `malloc()`, 这个函数在各种操作系统上的使用是一致的, `malloc()` 申请的内存的释放函数为 `free()`。

`malloc()` 的内存一定要被 `free()`, 否则会造成内存泄漏。理想情况下, `malloc()` 和 `free()` 应成对出现, 即谁申请, 就由谁释放。例如下面的一段程序:

```
char * function(void)
{
    char *p;
    p = (char *)malloc(...);
    if(p==NULL)
        ...;
    ... /* 一系列针对 p 的操作 */
    return p;
}
```

在某处调用 `function()`, 用完 `function()` 中动态申请的内存后将其 `free`, 如下:

```
char *q = function();
...
free(q);
```

上述代码并不合理的, 因为违反了 `malloc()` 和 `free()` 成对出现的原则。不满足这个原则, 会导致代码的耦合度增大, 因为用户在调用 `function()` 函数时需要知道其内部细节。

较好的做法是在调用处申请内存, 并传入 `function()` 函数, 如下所示:

```
char *p=malloc(...);
if(p==NULL)
    ...;
function(p);
...
free(p);
p=NULL;
```

而函数 `function()` 则接收参数 `p`, 如下所示:

```
void function(char *p)
{
    ... /* 一系列针对 p 的操作 */
}
```

完全让 `malloc()` 和 `free()` 成对出现有时候很难做到, 即便如此, 也应尽力将 `malloc()` 申请内存的释放限制在本模块范围之内。如果在 A 模块申请的内存需在 B 模块释放, 一般而言, 软件结构的设计可能存在问题。

对于 Linux 内核而言, C 库的 `malloc()` 函数通常通过 `brk()` 和 `mmap()` 两个系统调用来实现。

11.3.2 内核空间内存动态申请

在 Linux 内核空间申请内存涉及的函数主要包括 `kmalloc()`、`__get_free_pages()` 和 `vmalloc()` 等。`kmalloc()` 和 `__get_free_pages()`（及其类似函数）申请的内存位于物理内存映射区域，而且在物理上也是连续的，它们与真实的物理地址只有一个固定的偏移，因此存在较简单的转换关系。而 `vmalloc()` 在虚拟内存空间给出一块连续的内存区，实质上，这片连续的虚拟内存存在物理内存中并不一定连续，而 `vmalloc()` 申请的虚拟内存和物理内存之间也没有简单的换算关系。

1. `kmalloc()`

```
void *kmalloc(size_t size, int flags);
```

给 `kmalloc()` 的第一个参数是要分配的块的大小，第二个参数为分配标志，用于控制 `kmalloc()` 的行为。

最常用的分配标志是 `GFP_KERNEL`，其含义是在内核空间的进程中申请内存。`kmalloc()` 的底层依赖 `__get_free_pages()` 实现，分配标志的前缀 `GFP` 正好是这个底层函数的缩写。使用 `GFP_KERNEL` 标志申请内存时，若暂时不能满足，则进程会睡眠等待页，即会引起阻塞，因此不能在中断上下文或持有自旋锁的时候使用 `GFP_KERNEL` 申请内存。

在中断处理函数、`tasklet` 和内核定时器等非进程上下文中不能阻塞，此时驱动应当使用 `GFP_ATOMIC` 标志来申请内存。当使用 `GFP_ATOMIC` 标志申请内存时，若不存在空闲页，则不等待，直接返回。

其他的相对不常用的申请标志还包括 `GFP_USER`（用来为用户空间页分配内存，可能阻塞）、`GFP_HIGHUSER`（类似 `GFP_USER`，但是从高端内存分配）、`GFP_NOIO`（不允许任何 I/O 初始化）、`GFP_NOFS`（不允许进行任何文件系统调用）、`__GFP_DMA`（要求分配在能够 DMA 的内存区）、`__GFP_HIGHMEM`（指示分配的内存可以位于高端内存）、`__GFP_COLD`（请求一个较长时间不访问的页）、`__GFP_NOWARN`（当一个分配无法满足时，阻止内核发出警告）、`__GFP_HIGH`（高优先级请求，允许获得被内核保留给紧急状况使用的最后的内存页）、`__GFP_REPEAT`（分配失败则尽力重复尝试）、`__GFP_NOFAIL`（标志只许申请成功，不推荐）和 `__GFP_NORETRY`（若申请不到，则立即放弃）。

使用 `kmalloc()` 申请的内存应使用 `kfree()` 释放，这个函数的用法和用户空间的 `free()` 类似。

2. `__get_free_pages()`

`__get_free_pages()` 系列函数/宏是 Linux 内核本质上最底层的用于获取空闲内存的方法，因为底层的伙伴算法以 `page` 的 2^n 次幂为单位管理空闲内存，所以最底层的内存申请总是以页为单位的。

`__get_free_pages()` 系列函数/宏包括 `get_zeroed_page()`、`__get_free_page()` 和 `__get_free_pages()`。

```
get_zeroed_page(unsigned int flags);
```

该函数返回一个指向新页的指针并且将该页清零。

```
__get_free_page(unsigned int flags);
```

该宏返回一个指向新页的指针但是该页不清零，它实际上为：

```
#define __get_free_page(gfp_mask) \
    __get_free_pages((gfp_mask), 0)
```

就是调用了下面的 `__get_free_pages()` 申请 1 页。



```
_get_free_pages(unsigned int flags, unsigned int order);
```

该函数可分配多个页并返回分配内存的首地址,分配的页数为 2^{order} ,分配的页也不清零。`order` 允许的最大值是 10 (即 1 024 页) 或者 11 (即 2 048 页),依赖于具体的硬件平台。

`_get_free_pages()` 和 `get_zeroed_page()` 的实现中调用了 `alloc_pages()` 函数, `alloc_pages()` 既可以在内核空间分配,也可以在用户空间分配,其原型为:

```
struct page * alloc_pages(int gfp_mask, unsigned long order);
```

参数含义与 `_get_free_pages()` 类似,但它返回分配的第一个页的描述符而非首地址。

使用 `_get_free_pages()` 系列函数/宏申请的内存应使用下列函数释放:

```
void free_page(unsigned long addr);
```

```
void free_pages(unsigned long addr, unsigned long order);
```



如果申请和释放的 `order` 不一样,则会引起内存的混乱。

`_get_free_pages` 等函数在使用时,其申请标志的值与 `kmalloc()` 完全一样,各标志的含义也与 `kmalloc()` 完全一致,最常用的是 `GFP_KERNEL` 和 `GFP_ATOMIC`。

3. vmalloc()

`vmalloc()` 一般用在为只存在于软件中 (没有对应的硬件意义) 的较大的顺序缓冲区分配内存, `vmalloc()` 远大于 `_get_free_pages()` 的开销,为了完成 `vmalloc()`,新的页表需要被建立。因此,只是调用 `vmalloc()` 来分配少量的内存 (如 1 页) 是不妥的。

`vmalloc()` 申请的内存应使用 `vfree()` 释放, `vmalloc()` 和 `vfree()` 的函数原型如下:

```
void *vmalloc(unsigned long size);
```

```
void vfree(void * addr);
```

`vmalloc()` 不能用在原子上下文中,因为它的内部实现使用了标志为 `GFP_KERNEL` 的 `kmalloc()`。

使用 `vmalloc` 函数的一个例子函数是 `create_module()` 系统调用,它利用 `vmalloc()` 函数来获取被创建模块需要的内存空间。

4. slab 与内存池

一方面,完全使用页为单元申请和释放内存容易导致浪费 (如果要申请少量字节也需要 1 页); 另一方面,在操作系统的运作过程中,经常会涉及大量对象的重复生成、使用和释放内存问题。在 Linux 系统中所用到的对象,比较典型的例子是 `inode`、`task_struct` 等。如果我们能够用合适的方法使得在对象前后两次被使用时分配在同一块内存或同一类内存空间且保留了基本的数据结构,就可以大大提高效率。`slab` 算法就是针对上述特点设计的。实际上 `kmalloc()` 即是使用 `slab` 机制实现的。

(1) 创建 slab 缓存。

```
struct kmem_cache *kmem_cache_create(const char *name, size_t size,
    size_t align, unsigned long flags,
    void (*ctor)(void*, struct kmem_cache *, unsigned long),
    void (*dtor)(void*, struct kmem_cache *, unsigned long));
```

`kmem_cache_create()` 用于创建一个 `slab` 缓存,它是一个可以驻留任意数目全部同样大小的后备缓存。参数 `size` 是要分配的每个数据结构的大小,参数 `flags` 是控制如何进行分配的位掩码,包括 `SLAB_NO_REAP` (即使内存紧缺也不自动收缩这块缓存)、`SLAB_HWCACHE_ALIGN` (每个数据对象被对齐到一个缓存行)、`SLAB_CACHE_DMA` (要求数据对象在 DMA 内存区分配) 等。

(2) 分配 slab 缓存。

```
void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags);
```

上述函数在 kmem_cache_create() 创建的 slab 后备缓冲中分配一块并返回首地址指针。

(3) 释放 slab 缓存。

```
void kmem_cache_free(struct kmem_cache *cachep, void *objp);
```

上述函数释放由 kmem_cache_alloc() 分配的缓存。

(4) 收回 slab 缓存。

```
int kmem_cache_destroy(struct kmem_cache *cachep);
```

代码清单 11.2 给出了 slab 缓存的使用范例。

代码清单 11.2 slab 缓存使用范例

```
1 /*创建 slab 缓存*/
2 static kmem_cache_t *xxx_cachep;
3 xxx_cachep = kmem_cache_create("xxx", sizeof(struct xxx),
4                               0, SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL, NULL);
5 /*分配 slab 缓存*/
6 struct xxx *ctx;
7 ctx = kmem_cache_alloc(xxx_cachep, GFP_KERNEL);
8 .../* 使用 slab 缓存 */
9 /*释放 slab 缓存*/
10 kmem_cache_free(xxx_cachep, ctx);
11 kmem_cache_destroy(xxx_cachep);
```

在系统中通过/proc/slabinfo 结点可以获知当前 slab 的分配和使用情况，例如在 LDD6410 开发板上运行“cat /proc/slabinfo”：

```
# cat /proc/slabinfo | more
```

```
slabinfo - version: 2.1
# name          <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab>
: tunables <limit> <batchcount> <sharedfactor> : slabdata <active_slabs> <num_s
labs> <sharedavail>
rpc_buffers      8      8  2048    2    1 : tunables  24   12    0 : sla
bdata      4      4    0
rpc_tasks      8      24   160    24    1 : tunables 120   60    0 : sla
bdata      1      1    0
rpc_inode_cache 0      0   576     7    1 : tunables  54   27    0 : sla
bdata      0      0    0
flow_cache      0      0    0    80   48    1 : tunables 120   60    0 : sla
bdata      0      0    0
cfq_io_context  0      0    0    96   40    1 : tunables 120   60    0 : sla
bdata      0      0    0
cfq_queue       0      0    0    88   44    1 : tunables 120   60    0 : sla
bdata      0      0    0
mqueue_inode_cache 1      6   640     6    1 : tunables  54   27    0 : sl
abdata      1      1    0
jffs2_inode_cache 0      0    0    24  145    1 : tunables 120   60    0 : sla
bdata      0      0    0
jffs2_node_frag 0      0    0    24  145    1 : tunables 120   60    0 : sla
bdata      0      0    0
jffs2_refblock  0      0   248    16    1 : tunables 120   60    0 : sla
--More--
```

注意，slab 不是要代替 __get_free_pages()，其在最底层仍然依赖于 __get_free_pages()，slab 在底层每次申请 1 页或多页，之后再分隔这些页为更小的单元进行管理，从而节省了内存，也提高了 slab 缓冲对象的访问效率。

除了 slab 以外，在 Linux 内核中还包含对内存池的支持，内存池技术也是一种非常经典的用于分配大量小对象的后备缓存技术。



Linux 内核中, 与内存池相关的操作包括如下几种。

(1) 创建内存池。

```
mempool_t *mempool_create(int min_nr, mempool_alloc_t *alloc_fn,  
                           mempool_free_t *free_fn, void *pool_data);
```

`mempool_create()` 函数用于创建一个内存池, `min_nr` 参数是需要预分配对象的数目, `alloc_fn` 和 `free_fn` 是指向内存池机制提供的标准对象分配和回收函数的指针, 其原型分别为:

```
typedef void *(mempool_alloc_t)(int gfp_mask, void *pool_data);
```

和

```
typedef void (mempool_free_t)(void *element, void *pool_data);
```

`pool_data` 是分配和回收函数用到的指针, `gfp_mask` 是分配标记。只有当 `_GFP_WAIT` 标记被指定时, 分配函数才会休眠。

(2) 分配和回收对象。

在内存池中分配和回收对象需由以下函数来完成:

```
void *mempool_alloc(mempool_t *pool, int gfp_mask);  
void mempool_free(void *element, mempool_t *pool);
```

`mempool_alloc()` 用来分配对象, 如果内存池分配器无法提供内存, 那么就可以用预分配的池。

(3) 回收内存池。

```
void mempool_destroy(mempool_t *pool);
```

`mempool_create()` 函数创建的内存池需由 `mempool_destroy()` 来回收。

11.3.3 虚拟地址与物理地址关系

对于内核物理内存映射区的虚拟内存, 使用 `virt_to_phys()` 可以实现内核虚拟地址转化为物理地址, `virt_to_phys()` 的实现是体系结构相关的, 对于 ARM 而言, `virt_to_phys()` 的定义如代码清单 11.3 所示。

代码清单 11.3 `virt_to_phys()` 函数

```
1 static inline unsigned long virt_to_phys(void *x)  
2 {  
3     return __virt_to_phys((unsigned long)(x));  
4 }  
5 #define __virt_to_phys(x) ((x) - PAGE_OFFSET + PHYS_OFFSET)
```

上面转换过程的 `PAGE_OFFSET` 通常为 3GB, 而 `PHYS_OFFSET` 则定于为系统 DRAM 内存的基地址。因此, 对于 LDD6410 电路板而言, 并不是将 0 地址映射到 3GB, 而是将外接的 DDR SDRAM 的首地址映射到 3GB。

与之对应的函数为 `phys_to_virt()`, 它将物理地址转化为内核虚拟地址, `phys_to_virt()` 的定义如代码清单 11.4 所示。

代码清单 11.4 `phys_to_virt()` 函数

```
1 static inline void *phys_to_virt(unsigned long x)  
2 {  
3     return (void *) (__phys_to_virt((unsigned long)(x)));  
4 }  
5 #define __phys_to_virt(x) ((x) - PHYS_OFFSET + PAGE_OFFSET)
```

注意, 上述 `virt_to_phys()` 和 `phys_to_virt()` 方法仅适用于 896MB 以下的低端内存, 高端内存的

虚拟地址与物理地址之间不存在如此简单的换算关系。

11.4 设备 I/O 端口和 I/O 内存的访问

设备通常会提供一组寄存器来用于控制设备、读写设备和获取设备状态，即控制寄存器、数据寄存器和状态寄存器。这些寄存器可能位于 I/O 空间，也可能位于内存空间。当位于 I/O 空间时，通常被称为 I/O 端口，位于内存空间时，对应的内存空间被称为 I/O 内存。

11.4.1 Linux I/O 端口和 I/O 内存访问接口

1. I/O 端口

在 Linux 设备驱动中，应使用 Linux 内核提供的函数来访问定位于 I/O 空间的端口，这些函数包括如下几种。

(1) 读写字节端口（8 位宽）。

```
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
```

(2) 读写字端口（16 位宽）。

```
unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
```

(3) 读写长字端口（32 位宽）。

```
unsigned inl(unsigned port);
void outl(unsigned longword, unsigned port);
```

(4) 读写一串字节。

```
void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);
```

(5) insb() 从端口 port 开始读 count 个字节端口，并将读取结果写入 addr 指向的内存；outsb() 将 addr 指向的内存的 count 个字节连续地写入 port 开始的端口。

(6) 读写一串字。

```
void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);
```

(7) 读写一串长字。

```
void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);
```

上述各函数中 I/O 端口号 port 的类型高度依赖于具体的硬件平台，因此，只是写出了 unsigned。

2. I/O 内存

在内核中访问 I/O 内存之前，需首先使用 ioremap() 函数将设备所处的物理地址映射到虚拟地址。ioremap() 的原型如下：

```
void *ioremap(unsigned long offset, unsigned long size);
```

ioremap() 与 vmalloc() 类似，也需要建立新的页表，但是它并不进行 vmalloc() 中所执行的内存分配行为。ioremap() 返回一个特殊的虚拟地址，该地址可用来存取特定的物理地址范围。通过 ioremap() 获得的虚拟地址应该被 iounmap() 函数释放，其原型如下：

```
void iounmap(void * addr);
```



在设备的物理地址被映射到虚拟地址之后, 尽管可以直接通过指针访问这些地址, 但是可以使用 Linux 内核的如下一组函数来完成设备内存映射的虚拟地址的读写, 这些函数如下所示。

(1) 读 I/O 内存。

```
unsigned int ioread8(void *addr);
unsigned int ioread16(void *addr);
unsigned int ioread32(void *addr);
```

与上述函数对应的较早版本的函数为 (这些函数在 Linux 2.6 中仍然被支持):

```
unsigned readb(address);
unsigned readw(address);
unsigned readl(address);
```

(2) 写 I/O 内存。

```
void iowrite8(u8 value, void *addr);
void iowrite16(u16 value, void *addr);
void iowrite32(u32 value, void *addr);
```

与上述函数对应的较早版本的函数为 (这些函数在 Linux 2.6 中仍然被支持):

```
void writeb(unsigned value, address);
void writew(unsigned value, address);
void writel(unsigned value, address);
```

(3) 读一串 I/O 内存。

```
void ioread8_rep(void *addr, void *buf, unsigned long count);
void ioread16_rep(void *addr, void *buf, unsigned long count);
void ioread32_rep(void *addr, void *buf, unsigned long count);
```

(4) 写一串 I/O 内存。

```
void iowrite8_rep(void *addr, const void *buf, unsigned long count);
void iowrite16_rep(void *addr, const void *buf, unsigned long count);
void iowrite32_rep(void *addr, const void *buf, unsigned long count);
```

(5) 复制 I/O 内存。

```
void memcpy_fromio(void *dest, void *source, unsigned int count);
void memcpy_toio(void *dest, void *source, unsigned int count);
```

(6) 设置 I/O 内存。

```
void memset_io(void *addr, u8 value, unsigned int count);
```

3. 把 I/O 端口映射到内存空间

```
void *ioport_map(unsigned long port, unsigned int count);
```

通过这个函数, 可以把 port 开始的 count 个连续的 I/O 端口重映射为一段“内存空间”。然后就可以在其返回的地址上像访问 I/O 内存一样访问这些 I/O 端口。当不再需要这种映射时, 需要调用下面的函数来撤销。

```
void ioport_unmap(void *addr);
```

实际上, 分析 ioport_map() 的源代码可发现, 映射到内存空间行为实际上是给开发人员制造的一个“假象”, 并没有映射到内核虚拟地址, 仅仅是为了让工程师可使用统一的 I/O 内存访问接口访问 I/O 端口。

11.4.2 申请与释放设备 I/O 端口和 I/O 内存

1. I/O 端口申请

Linux 内核提供了一组函数用于申请和释放 I/O 端口。

```
struct resource *request_region(unsigned long first, unsigned long n, const char *name);
```

这个函数向内核申请 n 个端口，这些端口从 `first` 开始，`name` 参数为设备的名称。如果分配成功返回值是非 NULL，如果返回 NULL，则意味着申请端口失败。

当用 `request_region()` 申请的 I/O 端口使用完成后，应当使用 `release_region()` 函数将它们归还给系统，这个函数的原型如下：

```
void release_region(unsigned long start, unsigned long n);
```

2. I/O 内存申请

同样，Linux 内核也提供了一组函数用于申请和释放 I/O 内存的范围。

```
struct resource *request_mem_region(unsigned long start, unsigned long len, char *name);
```

这个函数向内核申请 n 个内存地址，这些地址从 `first` 开始，`name` 参数为设备的名称。如果分配成功返回值是非 NULL，如果返回 NULL，则意味着申请 I/O 内存失败。

当用 `request_mem_region()` 申请的 I/O 内存使用完成后，应当使用 `release_mem_region()` 函数将它们归还给系统，这个函数的原型如下：

```
void release_mem_region(unsigned long start, unsigned long len);
```

上述 `request_region()` 和 `release_mem_region()` 都不是必须的，但建议使用。其任务是检查申请的资源是否可用，如果可用则申请成功，并标志为已经使用，其他驱动想再次申请该资源时就会失败。

有许多设备驱动程序在没有申请 I/O 端口和 I/O 内存之前就直接访问了，这不够安全。

11.4.3 设备 I/O 端口和 I/O 内存访问流程

综合 11.3 节和本节的内容，可以归纳出设备驱动访问 I/O 端口和 I/O 内存的步骤。

I/O 端口访问的一种途径是直接使用 I/O 端口操作函数：在设备打开或驱动模块被加载时申请 I/O 端口区域，之后使用 `inb()`、`outb()` 等进行端口访问，最后，在设备关闭或驱动被卸载时释放 I/O 端口范围。整个流程如图 11.7 所示。

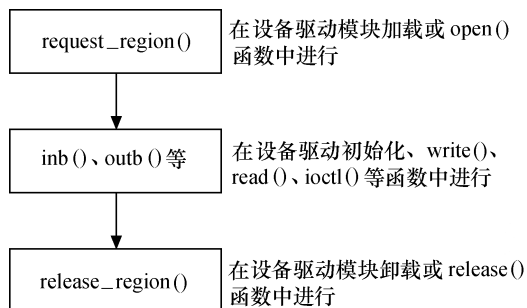


图 11.7 I/O 端口的访问流程 (不映射到内存空间)

I/O 端口访问的另一种途径是将 I/O 端口映射为内存进行访问：在设备打开或驱动模块被加载时，申请 I/O 端口区域并使用 `ioport_map()` 映射到内存，之后使用 I/O 内存的函数进行端口访问，最后，在设备关闭或驱动被卸载时释放 I/O 端口并释放映射。整个流程如图 11.8 所示。

I/O 内存的访问步骤如图 11.9 所示，首先是调用 `request_mem_region()` 申请资源，接着将寄存器地址通过 `ioremap()` 映射到内核空间虚拟地址，之后就可以通过 Linux 设备访问编程接口访问这些设备的寄存器了。访问完成后，应对 `ioremap()` 申请的虚拟地址进行释放，并释放 `release_mem_region()` 申请的 I/O 内存资源。

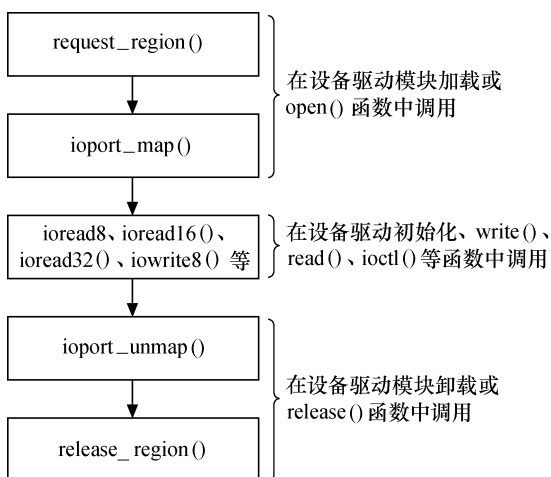


图 11.8 I/O 端口的访问流程 (映射到内存空间)

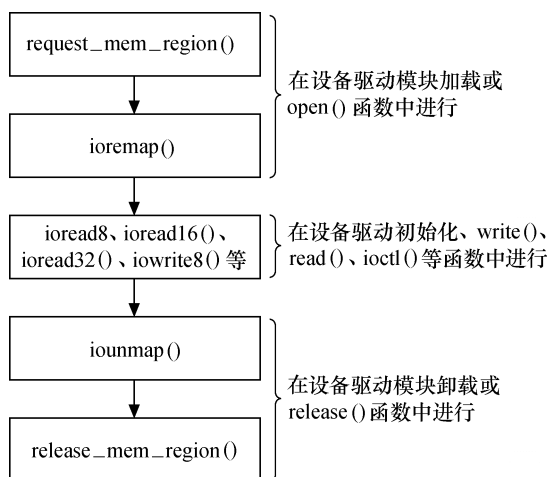


图 11.9 I/O 内存访问流程

11.4.4 将设备地址映射到用户空间

1. 内存映射与 VMA

一般情况下, 用户空间是不可能也不应该直接访问设备的, 但是, 设备驱动程序中可实现 `mmap()` 函数, 这个函数可使得用户空间直接访问设备的物理地址。实际上, `mmap()` 实现了这样的一个映射过程: 它将用户空间的一段内存与设备内存关联, 当用户访问用户空间的这段地址范围时, 实际上会转化为对设备的访问。

这种能力对于显示适配器一类的设备非常有意义, 如果用户空间可直接通过内存映射访问显存的话, 屏幕帧的各点的像素将不再需要一个从用户空间到内核空间的复制的过程。

`mmap()` 必须以 `PAGE_SIZE` 为单位进行映射, 实际上, 内存只能以页为单位进行映射, 若要映射非 `PAGE_SIZE` 整数倍的地址范围, 要先进行页对齐, 强行以 `PAGE_SIZE` 的倍数大小进行映射。

从 `file_operations` 文件操作结构体可以看出, 驱动中 `mmap()` 函数的原型如下:

```
int(*mmap)(struct file *, struct vm_area_struct*);
```

驱动中的 `mmap()` 函数将在用户进行 `mmap()` 系统调用时最终被调用, `mmap()` 系统调用的原型与 `file_operations` 中 `mmap()` 的原型区别很大, 如下所示:

```
caddr_t mmap (caddr_t addr, size_t len, int prot, int flags, int fd, off_t offset);
```

参数 `fd` 为文件描述符, 一般由 `open()` 返回, `fd` 也可以指定为 `-1`, 此时需指定 `flags` 参数中的 `MAP_ANON`, 表明进行的是匿名映射。

`len` 是映射到调用用户空间的字节数, 它从被映射文件开头 `offset` 个字节开始算起, `offset` 参数一般设为 `0`, 表示从文件头开始映射。

`prot` 参数指定访问权限, 可取如下几个值的“或”: `PROT_READ` (可读)、`PROT_WRITE` (可写)、`PROT_EXEC` (可执行) 和 `PROT_NONE` (不可访问)。

参数 `addr` 指定文件应被映射到用户空间的起始地址, 一般被指定为 `NULL`, 这样, 选择起始地址的任务将由内核完成, 而函数的返回值就是映射到用户空间的地址。其类型 `caddr_t` 实际上就是 `void *`。

当用户调用 `mmap()` 的时候，内核会进行如下处理。

- ① 在进程的虚拟空间查找一块 VMA。
- ② 将这块 VMA 进行映射。
- ③ 如果设备驱动程序或者文件系统的 `file_operations` 定义了 `mmap()` 操作，则调用它。
- ④ 将这个 VMA 插入进程的 VMA 链表中。

`file_operations` 中 `mmap()` 函数的第一个参数就是步骤①中找到的 VMA。

由 `mmap()` 系统调用映射的内存可由 `munmap()` 解除映射，这个函数的原型如下：

```
int munmap(caddr_t addr, size_t len);
```

驱动程序中 `mmap()` 的实现机制是建立页表，并填充 VMA 结构体中 `vm_operations_struct` 指针。

VMA 即 `vm_area_struct`，用于描述一个虚拟内存区域，VMA 结构体的定义如代码清单 11.5 所示。

代码清单 11.5 VMA 结构体

```
1 struct vm_area_struct {
2     struct mm_struct *vm_mm; /* 所处的地址空间 */
3     unsigned long vm_start; /* 开始虚拟地址 */
4     unsigned long vm_end; /* 结束虚拟地址 */
5
6     pgprot_t vm_page_prot; /* 访问权限 */
7     unsigned long vm_flags; /* 标志, VM_READ/VM_WRITE/VM_EXEC/VM_SHARED 等 */
8     ...
9     /* 操作 VMA 的函数集指针 */
10    struct vm_operations_struct *vm_ops;
11
12    unsigned long vm_pgoff; /* 偏移 (页帧号) */
13    struct file *vm_file;
14    void *vm_private_data;
15    ...
16};
```

VMA 结构体描述的虚地址介于 `vm_start` 和 `vm_end` 之间，而其 `vm_ops` 成员指向这个 VMA 的操作集。针对 VMA 的操作都被包含在 `vm_operations_struct` 结构体中，`vm_operations_struct` 结构体的定义如代码清单 11.6 所示。

代码清单 11.6 `vm_operations_struct` 结构体

```
1 struct vm_operations_struct {
2     void (*open)(struct vm_area_struct *area); /* 打开 VMA 的函数 */
3     void (*close)(struct vm_area_struct *area); /* 关闭 VMA 的函数 */
4     struct page *(*nopage)(struct vm_area_struct *area, unsigned long address,
5         int *type); /* 访问的页不在内存时调用 */
6     int (*populate)(struct vm_area_struct *area, unsigned long address, unsigned
7         long len, pgprot_t prot, unsigned long pgoff, int nonblock);
8     ...
9};
```

在内核生成一个 VMA 后，它会调用该 VMA 的 `open()` 函数，例如 `fork` 一个继承父继承资源的子进程时。但是，当用户进行 `mmap()` 系统调用后，尽管 VMA 在设备驱动文件操作结构体的 `mmap()` 被调用前就已产生，内核却不会调用 VMA 的 `open()` 函数，通常需要在驱动的 `mmap()` 函数中显示调用 `vma->vm_ops->open()`。代码清单 11.7 给出了一个 `vm_operations_struct` 的操作范例。



代码清单 11.7 vm_operations_struct 操作范例

```
1 static int xxx_mmap(struct file *filp, struct vm_area_struct *vma)
2 {
3     if (remap_pfn_range(vma, vma->vm_start, vm->vm_pgoff, vma->vm_end - vma
4         ->vm_start, vma->vm_page_prot)) /* 建立页表 */
5         return - EAGAIN;
6     vma->vm_ops = &xxx_remap_vm_ops;
7     xxx_vma_open(vma);
8     return 0;
9 }
10
11 void xxx_vma_open(struct vm_area_struct *vma) /* VMA 打开函数 */
12 {
13     ...
14     printk(KERN_NOTICE "xxx VMA open, virt %lx, phys %lx\n", vma->vm_start,
15         vma->vm_pgoff << PAGE_SHIFT);
16 }
17
18 void xxx_vma_close(struct vm_area_struct *vma) //VMA 关闭函数
19 {
20     ...
21     printk(KERN_NOTICE "xxx VMA close.\n");
22 }
23
24 static struct vm_operations_struct xxx_remap_vm_ops = { /* VMA 操作结构体 */
25     .open = xxx_vma_open,
26     .close = xxx_vma_close,
27     ...
28 };
```

第3行调用的 `remap_pfn_range()` 创建页表, 以 VMA 结构体的成员 (VMA 的数据成员是内核根据用户的请求自己填充的) 作为 `remap_pfn_range()` 的参数, 映射的虚拟地址范围是 `vma->vm_start` 至 `vma->vm_end`。

`remap_pfn_range()` 函数的原型如下:

```
int remap_pfn_range(struct vm_area_struct *vma, unsigned long addr,
    unsigned long pfn, unsigned long size, pgprot_t prot);
```

其中的 `addr` 参数表示内存映射开始处的虚拟地址。`remap_pfn_range()` 函数为 `addr~addr+size` 之间的虚拟地址构造页表。

`pfn` 是虚拟地址应该映射到的物理地址的页帧号, 实际上就是物理地址右移 `PAGE_SHIFT` 位。若 `PAGE_SIZE` 为 4KB, 则 `PAGE_SHIFT` 为 12, 因为 `PAGE_SIZE` 等于 $1 \ll \text{PAGE_SHIFT}$ 。

`prot` 是新页所要求的保护属性。

在驱动程序中, 我们能使用 `remap_pfn_range()` 映射内存中的保留页 (如 X86 系统中的 640KB~1MB 区域) 和设备 I/O 内存, 另外, `kmalloc()` 申请的内存若要被映射到用户空间可以通过 `mem_map_reserve()` 设置为保留后进行。代码清单 11.8 给出了映射 `kmalloc()` 申请的内存到用户空间的典型范例。

代码清单 11.8 映射 `kmalloc()` 申请的内存到用户空间范例

```
1 /*内核模块加载函数*/
2 int _ _init kmalloc_map_init(void)
3 {
4     ...
```

```

5  /* 申请设备号、添加 cdev 结构体 */
6  buffer = kmalloc(BUF_SIZE, GFP_KERNEL); //申请 buffer
7
8  for (page = virt_to_page(buffer); page < virt_to_page(buffer + BUF_SIZE);
9      page++)
10     mem_map_reserve(page); /* 置页为保留 */
11 }
12 /*mmap()函数*/
13 static int kmalloc_map_mmap(struct file *filp, struct vm_area_struct *vma)
14 {
15     unsigned long page, pos;
16     unsigned long start = (unsigned long)vma->vm_start;
17     unsigned long size = (unsigned long)(vma->vm_end - vma->vm_start);
18     printk(KERN_INFO "mmmaptest_mmap called\n");
19     /* 用户要映射的区域太大 */
20     if (size > BUF_SIZE)
21         return -EINVAL;
22
23     pos = (unsigned long)buffer;
24     /* 映射 buffer 中的所有页 */
25     while (size > 0) {
26         /* 每次映射一页 */
27         page = virt_to_phys((void*)pos);
28         if (remap_page_range(start, page, PAGE_SIZE, PAGE_SHARED))
29             return -EAGAIN;
30         start += PAGE_SIZE;
31         pos += PAGE_SIZE;
32         size -= PAGE_SIZE;
33     } return 0;
34 }

```

第 28 行调用 `remap_page_range(start, page, PAGE_SIZE, PAGE_SHARED)` 的第 4 个参数 `PAGE_SHARED` 实际上是 `_PAGE_PRESENT | _PAGE_USER | _PAGE_RW`，表明可读写并映射到用户空间。

通常，I/O 内存被映射时需要是 `nocache` 的，这时候，我们应该对 `vma->vm_page_prot` 设置 `nocache` 标志之后再映射，如代码清单 11.9 所示。

代码清单 11.9 以 `nocache` 方式将内核空间映射到用户空间

```

1 static int xxx_nocache_mmap(struct file *filp, struct vm_area_struct *vma)
2 {
3     vma->vm_page_prot = pgprot_noncached(vma->vm_page_prot); /* 赋 nocache 标志 */
4     vma->vm_pgoff = ((u32)map_start >> PAGE_SHIFT);
5     /* 映射 */
6     if (remap_pfn_range(vma, vma->vm_start, vma->vm_pgoff, vma->vm_end - vma
7         ->vm_start, vma->vm_page_prot))
8         return -EAGAIN;
9     return 0;
10 }

```

上述代码第 3 行的 `pgprot_noncached()` 是一个宏，它高度依赖于 CPU 体系结构，ARM 的 `pgprot_noncached()` 定义如下：

```

#define pgprot_noncached(prot) __pgprot(pgprot_val(prot) & ~(L_PTE_CACHEABLE | L_PTE_BUFFERABLE))

```

另一个比 `pgprot_noncached()` 稍微少一些限制的宏是 `pgprot_writecombine()`，它的定义如下：



```
#define pgprot_writecombine(prot) __pgprot(pgprot_val(prot) & ~L_PTE_CACHEABLE)
```

`pgprot_noncached()` 实际禁止了相关页的 Cache 和写缓冲 (write buffer), `pgprot_writecombine()` 则没有禁止写缓冲。ARM 的写缓冲器是一个非常小的 FIFO 存储器, 位于处理器核与主存之间, 其目的在于将处理器核和 Cache 从较慢的主存写操作中解脱出来。写缓冲区与 Cache 在存储层次上处于同一层次, 但是它只作用于写主存。

2. `nopage()` 函数

除了 `remap_pfn_range()` 以外, 在驱动程序中实现 VMA 的 `nopage()` 函数通常可以为设备提供更加灵活的内存映射途径。当访问的页不在内存, 即发生缺页异常时, `nopage()` 会被内核自动调用。这是因为, 当发生缺页异常时, 系统会经过如下处理过程。

- (1) 找到缺页的虚拟地址所在的 VMA。
- (2) 如果必要, 分配中间页目录表和页表。
- (3) 如果页表项对应的物理页面不存在, 则调用这个 VMA 的 `nopage()` 方法, 它返回物理页面的页描述符。
- (4) 将物理页面的地址填充到页表中。

实现 `nopage()` 后, 用户空间可以通过 `mremap()` 系统调用重新绑定映射区域所绑定的地址, 代码清单 11.10 给出了一个设备驱动中使用 `nopage()` 的典型范例。

代码清单 11.10 `nopage()` 函数使用范例

```
1 static int xxx_mmap(struct file *filp, struct vm_area_struct *vma)
2 {
3     unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
4     if (offset >= _pa(high_memory) || (filp->f_flags & O_SYNC))
5         vma->vm_flags |= VM_IO;
6     vma->vm_flags |= VM_RESERVED; /* 预留 */
7     vma->vm_ops = &xxx_nopage_vm_ops;
8     xxx_vma_open(vma);
9     return 0;
10 }
11
12 struct page *xxx_vma_nopage(struct vm_area_struct *vma, unsigned long
13     address, int *type)
14 {
15     struct page *pageptr;
16     unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
17     unsigned long physaddr = address - vma->vm_start + offset; /* 物理地址 */
18     unsigned long pageframe = physaddr >> PAGE_SHIFT; /* 页帧号 */
19     if (!pfn_valid(pageframe)) /* 页帧号有效? */
20         return NOPAGE_SIGBUS;
21     pageptr = pfn_to_page(pageframe); /* 页帧号->页描述符 */
22     get_page(pageptr); /* 获得页, 增加页的使用计数 */
23     if (type)
24         *type = VM_FAULT_MINOR;
25     return pageptr; /* 返回页描述符 */
26 }
```

上述函数对常规内存进行映射, 返回一个页描述符, 可用于扩大或缩小映射的内存区域。

由此可见, `nopage()` 与 `remap_pfn_range()` 的一个较大区别在于 `remap_pfn_range()` 一般用于设备内存映射, 而 `nopage()` 还可用于 RAM 映射, 其调用发生在缺页异常时。



大多数设备驱动都不需要提供设备内存到用户空间的映射能力，因为，对于串口等面向流的设备而言，实现这种映射毫无意义。而对于显示、视频等设备，建立映射可减少用户空间和内核空间之间的内存拷贝。

11.5 I/O 内存静态映射

在将 Linux 移植到目标电路板的过程中，通常会建立外设 I/O 内存物理地址到虚拟地址的静态映射，这个映射通过在电路板对应的 `map_desc` 结构体数组中添加新的成员来完成，`map_desc` 结构体的定义如代码清单 11.11 所示。

代码清单 11.11 `map_desc` 结构体

```
1 struct map_desc {
2     unsigned long virtual; /* 虚拟地址 */
3     unsigned long pfn ; /* __phys_to_pfn(phy_addr) */
4     unsigned long length; /* 大小 */
5     unsigned int type; /* 类型 */
6 };
```

例如，在内核 `arch/arm/mach-ixp2000/ixdp2x01.c` 文件对应的 Intel IXDP2401 和 IXDP2801 平台上包含一个 CPLD，该文件中就进行了 CPLD 物理地址到虚拟地址的静态映射，如代码清单 11.12 所示。

代码清单 11.12 在板文件中增加物理地址到虚拟地址的静态映射

```
1 static struct map_desc ixdp2x01_io_desc __initdata = {
2     .virtual      = IXDP2X01_VIRT_CPLD_BASE,
3     .pfn          = __phys_to_pfn(IXDP2X01_PHYS_CPLD_BASE),
4     .length       = IXDP2X01_CPLD_REGION_SIZE,
5     .type         = MT_DEVICE
6 };
7
8 static void __init ixdp2x01_map_io(void)
9 {
10     ixp2000_map_io();
11     iotable_init(&ixdp2x01_io_desc, 1);
12 }
```

代码清单 11.12 中的第 11 行 `iotable_init()` 是最终建立页映射的函数，它被通过 `MACHINE_START`、`MACHINE_END` 宏赋值给板的 `map_io()` 函数。Linux 操作系统移植到特定平台上，`MACHINE_START`、`MACHINE_END` 宏之间的定义针对特定电路板而设计，其中的 `map_io()` 成员函数完成 I/O 内存的静态映射，代码清单 11.13 给出了 IXDP2401 电路板的 `MACHINE_START`、`MACHINE_END` 宏的例子。

代码清单 11.13 IXDP2401 电路板的 `MACHINE_START`、`MACHINE_END` 宏

```
1 MACHINE_START(IXDP2401, "Intel IXDP2401 Development Platform")
2     /* Maintainer: MontaVista Software, Inc. */
3     .phys_io      = IXP2000_UART_PHYS_BASE,
```



```
4     .io_pg_offst   = ((IXP2000_UART_VIRT_BASE) >> 18) & 0xfffc,
5     .boot_params   = 0x00000100,
6     .map_io        = ixdp2x01_map_io,
7     .init_irq      = ixdp2x01_init_irq,
8     .timer         = &ixdp2x01_timer,
9     .init_machine  = ixdp2x01_init_machine,
10 MACHINE_END
```

在一个已经移植好 OS 的内核中，驱动工程师完全可以对非常规内存区域的 I/O 内存（外设控制器寄存器、MCU 内部集成的外设控制器寄存器等）依照电路板的资源使用情况添加到 `map_desc` 数组中，代码清单 11.14 的例子给出了内存空间资源的使用情况（注释部分）与 `map_desc` 数组的对应关系。

代码清单 11.14 根据电路板内存资源情况定义 `map_desc`

```
1  /*
2  * 逻辑地址 物理地址
3  * e8000000    40000000 PCI memory          PHYS_PCI_MEM_BASE (max 512M)
4  * ec000000    61000000 PCI 配置空间      PHYS_PCI_CONFIG_BASE (max 16M)
5  * ed000000    62000000 PCI V3 regs       PHYS_PCI_V3_BASE (max 64k)
6  * ee000000    60000000 PCI IO            PHYS_PCI_IO_BASE (max 16M)
7  * ef000000    Cache flush
8  * f1000000    10000000 核心模块寄存器
9  * f1100000    11000000 系统控制寄存器
10 * f1200000    12000000 EBI 寄存器
11 * f1300000    13000000 计数器/定时器
12 * f1400000    14000000 中断控制器
13 * f1600000    16000000 UART 0
14 * f1700000    17000000 UART 1
15 * f1a00000    1a000000 调试用 LEDs
16 * f1b00000    1b000000 GPIO
17 */
18
19 static struct map_desc ap_io_desc[] __initdata = {
20 {
21     .virtual = IO_ADDRESS (INTEGRATOR_HDR_BASE),
22     .pfn     = __phys_to_pfn (INTEGRATOR_HDR_BASE),
23     .length  = SZ_4K,
24     .type    = MT_DEVICE
25 }, {
26     .virtual = IO_ADDRESS (INTEGRATOR_SC_BASE),
27     .pfn     = __phys_to_pfn (INTEGRATOR_SC_BASE),
28     .length  = SZ_4K,
29     .type    = MT_DEVICE
30 }, {
31     .virtual = IO_ADDRESS (INTEGRATOR_EBI_BASE),
32     .pfn     = __phys_to_pfn (INTEGRATOR_EBI_BASE),
33     .length  = SZ_4K,
34     .type    = MT_DEVICE
35 }, {
36     .virtual = IO_ADDRESS (INTEGRATOR_CT_BASE),
37     .pfn     = __phys_to_pfn (INTEGRATOR_CT_BASE),
38     .length  = SZ_4K,
39     .type    = MT_DEVICE
40 }, {
```

```

41     .virtual = IO_ADDRESS (INTEGRATOR_IC_BASE),
42     .pfn     = __phys_to_pfn (INTEGRATOR_IC_BASE),
43     .length  = SZ_4K,
44     .type    = MT_DEVICE
45 }, {
46     .virtual = IO_ADDRESS (INTEGRATOR_UART0_BASE),
47     .pfn     = __phys_to_pfn (INTEGRATOR_UART0_BASE),
48     .length  = SZ_4K,
49     .type    = MT_DEVICE
50 }, {
51     .virtual = IO_ADDRESS (INTEGRATOR_UART1_BASE),
52     .pfn     = __phys_to_pfn (INTEGRATOR_UART1_BASE),
53     .length  = SZ_4K,
54     .type    = MT_DEVICE
55 }, {
56     .virtual = IO_ADDRESS (INTEGRATOR_DBG_BASE),
57     .pfn     = __phys_to_pfn (INTEGRATOR_DBG_BASE),
58     .length  = SZ_4K,
59     .type    = MT_DEVICE
60 }, {
61     .virtual = IO_ADDRESS (INTEGRATOR_GPIO_BASE),
62     .pfn     = __phys_to_pfn (INTEGRATOR_GPIO_BASE),
63     .length  = SZ_4K,
64     .type    = MT_DEVICE
65 }, {
66     .virtual = PCI_MEMORY_VADDR,
67     .pfn     = __phys_to_pfn (PHYS_PCI_MEM_BASE),
68     .length  = SZ_16M,
69     .type    = MT_DEVICE
70 }, {
71     .virtual = PCI_CONFIG_VADDR,
72     .pfn     = __phys_to_pfn (PHYS_PCI_CONFIG_BASE),
73     .length  = SZ_16M,
74     .type    = MT_DEVICE
75 }, {
76     .virtual = PCI_V3_VADDR,
77     .pfn     = __phys_to_pfn (PHYS_PCI_V3_BASE),
78     .length  = SZ_64K,
79     .type    = MT_DEVICE
80 }, {
81     .virtual = PCI_IO_VADDR,
82     .pfn     = __phys_to_pfn (PHYS_PCI_IO_BASE),
83     .length  = SZ_64K,
84     .type    = MT_DEVICE
85 }
86 };

```

此后，在设备驱动中访问经过 `map_desc` 数组映射后的 I/O 内存时，直接在 `map_desc` 中该段的虚拟地址上加上相应的偏移即可，不再需要使用 `ioremap()`。

我们若要在 LDD6410 开发板的板文件中添加新的物理地址到虚拟地址的映射，只需要修改文件 `arch/arm/mach-s3c6410/mach-ldd6410.c` 中的 `map_desc` 数组（目前该数组为空）：

```
struct map_desc ldd6410_iodesc[] = {};
```



11.6 DMA

DMA 是一种无需 CPU 的参与就可以让外设与系统内存之间进行双向数据传输的硬件机制。使用 DMA 可以使系统 CPU 从实际的 I/O 数据传输过程中摆脱出来, 从而大大提高系统的吞吐量。DMA 通常与硬件体系结构特别是外设的总线技术密切相关。

DMA 方式的数据传输由 DMA 控制器 (DMAC) 控制, 在传输期间, CPU 可以并发地执行其他任务。当 DMA 结束后, DMAC 通过中断通知 CPU 数据传输已经结束, 然后由 CPU 执行相应的中断服务程序进行后处理。

11.6.1 DMA 与 Cache 一致性

Cache 和 DMA 本身似乎是两个毫不相关的事物。Cache 被用做 CPU 针对内存的缓存, 利用程序的空间局部性和时间局部性原理, 达到较高的命中率从而避免 CPU 每次都一定要与相对慢速的内存交互数据来提高数据的访问速率。DMA 可以用做内存与外设之间传输数据的方式, 这种传输方式之下, 数据并不需要经过 CPU 中转。

假设 DMA 针对内存的目的地址与 Cache 缓存的对象没有重叠区域 (如图 11.10 所示), DMA 和 Cache 之间将相安无事。但是, 如果 DMA 的目的地址与 Cache 所缓存的内存地址访问有重叠 (如图 11.11 所示), 经过 DMA 操作, Cache 缓存对应的内存的数据已经被修改, 而 CPU 本身并不知道, 它仍然认为 Cache 中的数据就是内存中的数据, 以后访问 Cache 映射的内存时, 它仍然使用陈旧的 Cache 数据。这样就发生 Cache 与内存之间数据“不一致性”的错误。

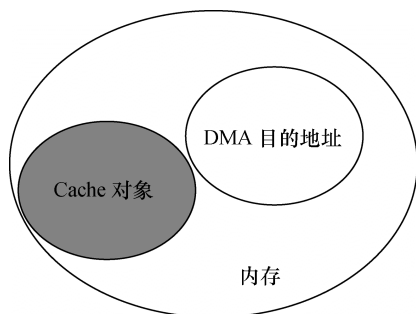


图 11.10 DMA 目的地址与 Cache 对象不交叉

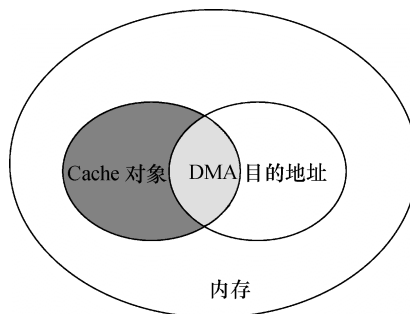


图 11.11 DMA 目的地址与 Cache 对象交叉

所谓 Cache 数据与内存数据的不一致性, 是指在采用 Cache 的系统中, 同样一个数据可能既存在于 Cache 中, 也存在于主存中, Cache 与主存中的数据一样则具有一致性, 数据若不一样则具有不一致性。

需要特别注意的是, Cache 与内存的一致性问题经常被初学者遗忘。在发生 Cache 与内存不一致性错误后, 驱动将无法正常运行。如果没有相关的背景知识, 工程师几乎无法定位错误的原因, 因为看起来所有的程序都是完全正确的。

解决由于 DMA 导致的 Cache 一致性问题的最简单方法是直接禁止 DMA 目标地址范围内内存的 Cache 功能。当然, 这将牺牲性能, 但是却更可靠, 图 11.12 所示为 Cache 和 DMA 在考虑性能和

易用两个方面时处于跷跷板两端的比重。

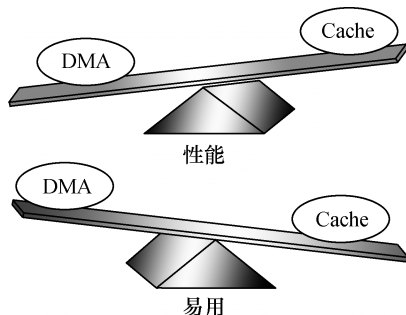


图 11.12 跷跷板两端的 Cache 与 DMA



Cache 的不一致问题并非只是发生在 DMA 的情况下，实际上，还存在于 Cache 使能和关闭的时刻。例如，对于带 MMU 功能的 ARM 处理器，在开启 MMU 之前，需要先置 Cache 无效，TLB 也是如此，代码清单 11.15 所给出的一段汇编被用来完成此任务。

代码清单 11.15 置 ARM 的 Cache 无效

```
1 /* 使 cache 无效 */
2 "mov    r0, #0\n"
3 "mcr    p15, 0, r0, c7, c7, 0\n" /* 使数据和指令 cache 无效 */
4 "mcr    p15, 0, r0, c7, c10, 4\n" /* 放空写缓冲 */
5 "mcr    p15, 0, r0, c8, c7, 0\n" /* 使 TLB 无效 */
```

11.6.2 Linux 下的 DMA 编程

首先 DMA 本身不属于一种等同于字符设备、块设备和网络设备的外设，它只是外设与内存交互数据的一种方式。因此，本节的标题不是“Linux 下的 DMA 驱动”而是“Linux 下的 DMA 编程”。

内存中用于与外设交互数据的一块区域被称做 DMA 缓冲区，在设备不支持 scatter/gather（分散/聚集，简称 SG）操作的情况下，DMA 缓冲区必须是物理上连续的。

1. DMA ZONE

对于 X86 系统的 ISA 设备而言，其 DMA 操作只能在 16MB 以下的内存中进行，因此，在使用 `kmalloc()` 和 `__get_free_pages()` 及其类似函数申请 DMA 缓冲区时应使用 `GFP_DMA` 标志，这样能保证获得的内存位于 `DMA_ZONE`，是具备 DMA 能力的。

内核中定义了 `__get_free_pages()` 针对 DMA 的“快捷方式” `__get_dma_pages()`，它在申请标志中添加了 `GFP_DMA`，如下所示：

```
#define __get_dma_pages(gfp_mask, order) \
    __get_free_pages((gfp_mask) | GFP_DMA, (order))
```

如果不想使用 `log2size` 即 `order` 为参数申请 DMA 内存，则可以使用另一个函数 `dma_mem_alloc()`，其源代码如代码清单 11.16 所示。

代码清单 11.16 `dma_mem_alloc()` 函数

```
1 static unsigned long dma_mem_alloc(int size)
2 {
```



```
3 int order = get_order(size); /* 大小->指数 */
4 return __get_dma_pages(GFP_KERNEL, order);
5 }
```

对于大多数现代嵌入式处理器而言, DMA 操作可以在整个常规内存区域进行, 因此 DMA ZONE 就直接覆盖了常规内存。

2. 虚拟地址, 物理地址和总线地址

基于 DMA 的硬件使用总线地址而非物理地址, 总线地址是从设备角度上看到的内存地址, 物理地址则是从 CPU MMU 控制器外围角度上看到的内存地址 (从 CPU 核角度看到的是虚拟地址)。虽然在 PC 上, 对于 ISA 和 PCI 而言, 总线地址即为物理地址, 但并非每个平台都是如此。因为有时候接口总线通过桥接电路被连接, 桥接电路会将 I/O 地址映射为不同的物理地址。例如, 在 PReP (PowerPC Reference Platform) 系统中, 物理地址 0 在设备端看起来是 0x80000000, 而 0 通常又被映射为虚拟地址 0xC0000000, 所以同一地址就具备了三重身份: 物理地址 0、总线地址 0x80000000 及虚拟地址 0xC0000000。还有一些系统提供了页面映射机制, 它能将任意的页面映射为连续的外设总线地址。内核提供了如下函数用于进行简单的虚拟地址/总线地址转换:

```
unsigned long virt_to_bus(volatile void *address);
void *bus_to_virt(unsigned long address);
```

在使用 IOMMU 或反弹缓冲区的情况下, 上述函数一般不会正常工作。而且, 这两个函数并不建议使用。如图 11.13 所示, IOMMU 的工作原理与 CPU 内的 MMU 非常类似, 不过它针对的是外设总线地址和内存地址之间的转化。由于 IOMMU 可以使得外设 DMA 引擎看到“虚拟地址”, 因此在使用 IOMMU 的情况下, 在修改映射寄存器后, 可以使得 SG 中分段的缓冲区地址对外设变得连续。

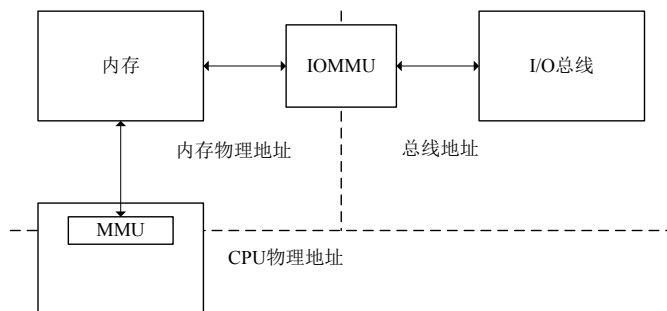


图 11.13 MMU 与 IOMMU

3. DMA 地址掩码

设备并不一定能在所有的内存地址上执行 DMA 操作, 在这种情况下应该通过下列函数执行 DMA 地址掩码:

```
int dma_set_mask(struct device *dev, u64 mask);
```

例如, 对于只能在 24 位地址上执行 DMA 操作的设备而言, 就应该调用 `dma_set_mask (dev, 0xfffffff)`。

4. 一致性 DMA 缓冲区

DMA 映射包括两个方面的工作: 分配一片 DMA 缓冲区; 为这片缓冲区产生设备可访问的地址。同时, DMA 映射也必须考虑 Cache 一致性问题。内核中提供了以下函数用于分配一个 DMA 一致性的内存区域:

```
void * dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t *handle, gfp_t gfp);
```

上述函数的返回值为申请到的 DMA 缓冲区的虚拟地址，此外，该函数还通过参数 `handle` 返回 DMA 缓冲区的总线地址。`handle` 的类型为 `dma_addr_t`，代表的是总线地址。

`dma_alloc_coherent()` 申请一片 DMA 缓冲区，进行地址映射并保证该缓冲区的 Cache 一致性。与 `dma_alloc_coherent()` 对应的释放函数为：

```
void dma_free_coherent(struct device *dev, size_t size, void *cpu_addr, dma_addr_t handle);
```

以下函数用于分配一个写合并（writecombining）的 DMA 缓冲区：

```
void * dma_alloc_writecombine(struct device *dev, size_t size, dma_addr_t *handle, gfp_t gfp);
```

与 `dma_alloc_writecombine()` 对应的释放函数 `dma_free_writecombine()` 实际上就是 `dma_free_coherent()`，因为它定义为：

```
#define dma_free_writecombine(dev, size, cpu_addr, handle) \
    dma_free_coherent(dev, size, cpu_addr, handle)
```

此外，Linux 内核还提供了 PCI 设备申请 DMA 缓冲区的函数 `pci_alloc_consistent()`，其原型为：

```
void * pci_alloc_consistent(struct pci_dev *pdev, size_t size, dma_addr_t *dma_addrp);
```

对应的释放函数为 `pci_free_consistent()`，其原型为：

```
void pci_free_consistent(struct pci_dev *pdev, size_t size, void *cpu_addr,
dma_addr_t dma_addr);
```

5. 流式 DMA 缓冲区

并非所有的 DMA 缓冲区都是驱动申请的，如果是驱动申请的，用一致性 DMA 缓冲区自然最方便，直接考虑了 Cache 一致性问题。但是，许多情况下，缓冲区来自内核的较上层（如网卡驱动中的网络报文、块设备驱动中要写入设备的数据等），上层很可能用的是普通的 `kmalloc()`、`__get_free_pages()` 等方法申请，这时候就要使用流式 DMA 映射。流式 DMA 缓冲区使用的一般步骤如下。

- （1）进行流式 DMA 映射。
- （2）执行 DMA 操作。
- （3）进行流式 DMA 去映射。

流式 DMA 映射操作在本质上多数就是进行 Cache 的 `invalidate` 或 `flush` 操作，以解决 Cache 一致性问题。

相对于一致性 DMA 映射而言，流式 DMA 映射的接口较为复杂。对于单个已经分配的缓冲区而言，使用 `dma_map_single()` 可实现流式 DMA 映射，该函数原型为：

```
dma_addr_t dma_map_single(struct device *dev, void *buffer, size_t size,
enum dma_data_direction direction);
```

如果映射成功，返回的是总线地址，否则，返回 `NULL`。第 4 个参数为 DMA 的方向，可能的值包括 `DMA_TO_DEVICE`、`DMA_FROM_DEVICE`、`DMA_BIDIRECTIONAL` 和 `DMA_NONE`。

`dma_map_single()` 的反函数为 `dma_unmap_single()`，原型是：

```
void dma_unmap_single(struct device *dev, dma_addr_t dma_addr, size_t size,
enum dma_data_direction direction);
```

通常情况下，设备驱动不应该访问 `unmap` 的流式 DMA 缓冲区，如果一定要这么做，可先使用如下函数获得 DMA 缓冲区的拥有权：

```
void dma_sync_single_for_cpu(struct device *dev, dma_handle_t bus_addr,
size_t size, enum dma_data_direction direction);
```

在驱动访问完 DMA 缓冲区后，应该将其所有权返还给设备，通过如下函数完成：



```
void dma_sync_single_for_device(struct device *dev, dma_handle_t bus_addr,
                                size_t size, enum dma_data_direction direction);
```

如果设备要求较大的 DMA 缓冲区, 在其支持 SG 模式的情况下, 申请不连续的多个相对较小的 DMA 缓冲区通常是防止申请太大的连续物理空间的方法。在 Linux 内核中, 使用如下函数映射 SG:

```
int dma_map_sg(struct device *dev, struct scatterlist *sg, int nents,
               enum dma_data_direction direction);
```

`nents` 是散列表 (scatterlist) 入口的数量, 该函数的返回值是 DMA 缓冲区的数量, 可能小于 `nents`。对于 scatterlist 中的每个项目, `dma_map_sg()` 为设备产生恰当的总线地址, 它会合并物理上临近的内存区域。

scatterlist 结构体的定义如代码清单 11.17 所示, 它包含了 scatterlist 对应的 page 结构体指针、缓冲区在 page 中的偏移 (offset)、缓冲区长度 (length) 以及总线地址 (dma_address)。

代码清单 11.17 scatterlist 结构体

```
1 struct scatterlist {
2     struct page *page;
3     unsigned int offset;
4     dma_addr_t dma_address;
5     unsigned int length;
6 };
```

执行 `dma_map_sg()` 后, 通过 `sg_dma_address()` 可返回 scatterlist 对应缓冲区的总线地址, `sg_dma_len()` 可返回 scatterlist 对应缓冲区的长度, 这两个函数的原型为:

```
dma_addr_t sg_dma_address(struct scatterlist *sg);
unsigned int sg_dma_len(struct scatterlist *sg);
```

在 DMA 传输结束后, 可通过 `dma_map_sg()` 的反函数 `dma_unmap_sg()` 去除 DMA 映射:

```
void dma_unmap_sg(struct device *dev, struct scatterlist *list,
                  int nents, enum dma_data_direction direction);
```

SG 映射属于流式 DMA 映射, 与单一缓冲区情况下的流式 DMA 映射类似, 如果设备驱动一定要访问映射情况下的 SG 缓冲区, 应该先调用如下函数:

```
void dma_sync_sg_for_cpu(struct device *dev, struct scatterlist *sg,
                          int nents, enum dma_data_direction direction);
```

访问完后, 通过下列函数将所有权返回给设备:

```
void dma_sync_sg_for_device(struct device *dev, struct scatterlist *sg,
                             int nents, enum dma_data_direction direction);
```

Linux 系统中可以有一个相对简单的方法预先分配缓冲区, 那就是同步 “mem=” 参数预留内存。例如, 对于内存为 64MB 的系统, 通过给其传递 `mem=62MB` 命令行参数可以使得顶部的 2MB 内存被预留出来作为 I/O 内存使用, 这 2MB 内存可以被静态映射 (11.5 节), 也可以被执行 `ioremap()`。

6. 申请和释放 DMA 通道

和中断一样, 在使用 DMA 之前, 设备驱动程序需首先向系统申请 DMA 通道, 申请 DMA 通道的函数如下:

```
int request_dma(unsigned int dmanr, const char * device_id);
```

同样的, 设备结构体指针可作为传入 `device_id` 的最佳参数。

使用完 DMA 通道后, 应该利用如下函数释放该通道:

```
void free_dma(unsigned int dmanr);
```

现在可以总结出在 Linux 设备驱动中 DMA 相关代码的流程, 如图 11.14 所示。

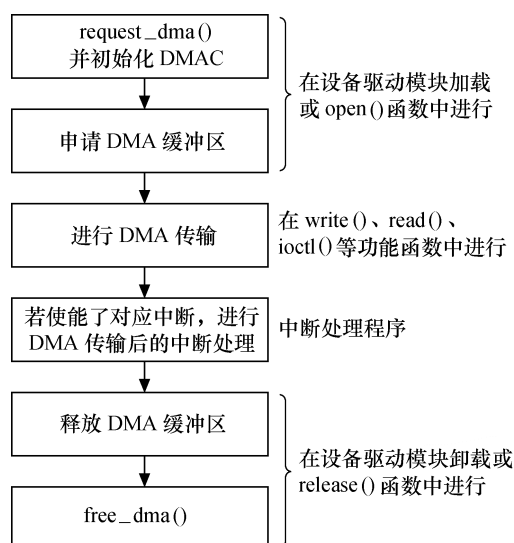


图 11.14 Linux 中 DMA 使用流程

11.7 总结

外设可处于 CPU 的内存空间和 I/O 空间，除 X86 外，嵌入式处理器一般只存在内存空间。在 Linux 系统中，为 I/O 内存和 I/O 端口的访问提高了一套统一的方法，访问流程一般为“申请资源→映射→访问→去映射→释放资源”。

对于有 MMU 的处理器而言，Linux 系统的内部布局比较复杂，可直接映射的物理内存称为常规内存，超出部分为高端内存。`kmalloc()`和 `_get_free_pages()`申请的内存存在物理上连续，而 `vmalloc()`申请的内存存在物理上不连续。

DMA 操作可能导致 Cache 的不一致问题，因此，对于 DMA 缓冲，应该使用 `dma_alloc_coherent()` 等方法申请。在 DMA 操作中涉及总线地址、物理地址和虚拟地址等概念，区分这 3 类地址非常重要。Linux 内核中对 DMA 通道的申请和释放采用了和中断类似的方法。

LINUX

第12章 工程中的 Linux 设备驱动

前面数章我们看到了 globalmem、globalfifo 这样类型的简单的字符设备驱动，但是，纵观 Linux 内核的源代码，读者都几乎找不到形式如此简单的驱动。

在实际的 Linux 驱动中，会看到一些其他数据结构、API 和设备驱动的一些新特性，因此，本章将带领您走入真实世界里的设备驱动。

12.1 全面介绍了 platform 设备和驱动，以及 platform 的意义。

12.2 节和 12.3 分别分析了 Linux 设备驱动的分层设计思想和主机与外设驱动分离的设计思想，并以输入设备、RTC 设备、SPI 主机和外设驱动进行了例证。

12.4 节介绍了 Linux 设备驱动的电源管理，suspend()和 resume()接口。

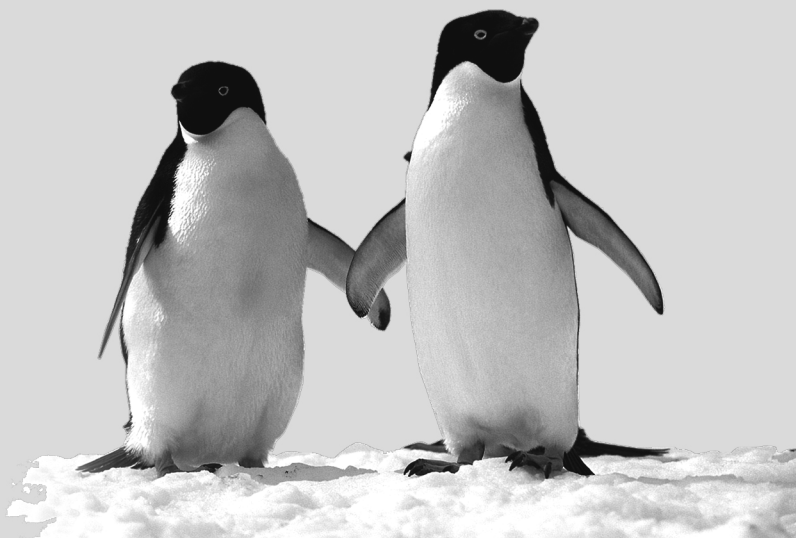
12.5 节介绍了混杂设备 miscdevice 驱动。

12.6 节介绍了基于 sysfs 的驱动。

12.7 节讲解了设备驱动中加载 firmware 的过程。

12.8 节对 Android 的驱动以及 Android 引入的内核补丁进行了介绍。

上述各节中的内容都与工程实际相关，各节之间是并列关系。



12.1 platform 设备驱动

12.1.1 platform 总线、设备与驱动

在 Linux 2.6 的设备驱动模型中，关心总线、设备和驱动这 3 个实体，总线将设备和驱动绑定。在系统每注册一个设备的时候，会寻找与之匹配的驱动；相反的，在系统每注册一个驱动的时候，会寻找与之匹配的设备，而匹配由总线完成。

一个现实的 Linux 设备和驱动通常都需要挂接在一种总线上，对于本身依附于 PCI、USB、I²C、SPI 等的设备而言，这自然不是问题，但是在嵌入式系统里面，SoC 系统中集成的独立的外设控制器、挂接在 SoC 内存空间的外设等确不依附于此类总线。基于这一背景，Linux 发明了一种虚拟的总线，称为 platform 总线，相应的设备称为 platform_device，而驱动成为 platform_driver。

注意，所谓的 platform_device 并不是与字符设备、块设备和网络设备并列的概念，而是 Linux 系统提供的一种附加手段，例如，在 S3C6410 处理器中，把内部集成的 I²C、RTC、SPI、LCD、看门狗等控制器都归纳为 platform_device，而它们本身就是字符设备。platform_device 结构体的定义如代码清单 12.1 所示。

代码清单 12.1 platform_device 结构体

```
1 struct platform_device {
2     const char    * name;           /* 设备名 */
3     u32          id;
4     struct device dev;
5     u32          num_resources;      /* 设备所使用各类资源数量 */
6     struct resource * resource;      /* 资源 */
7 };
```

platform_driver 这个结构体中包含 probe()、remove()、shutdown()、suspend()、resume() 函数，通常也需要由驱动实现，如代码清单 12.2 所示。

代码清单 12.2 platform_driver 结构体

```
1 struct platform_driver {
2     int (*probe)(struct platform_device *);
3     int (*remove)(struct platform_device *);
4     void (*shutdown)(struct platform_device *);
5     int (*suspend)(struct platform_device *, pm_message_t state);
6     int (*suspend_late)(struct platform_device *, pm_message_t state);
7     int (*resume_early)(struct platform_device *);
8     int (*resume)(struct platform_device *);
9     struct pm_ext_ops *pm;
10    struct device_driver driver;
11};
```

系统中为 platform 总线定义了一个 bus_type 的实例 platform_bus_type，其定义如代码清单 12.3 所示。

代码清单 12.3 platform 总线的 bus_type 实例 platform_bus_type

```
1 struct bus_type platform_bus_type = {
2     .name          = "platform",
3     .dev_attrs      = platform_dev_attrs,
```



```
4     .match      = platform_match,  
5     .uevent     = platform_uevent,  
6     .pm         = PLATFORM_PM_OPS_PTR,  
7 };  
8 EXPORT_SYMBOL_GPL(platform_bus_type);
```

这里要重点关注其 `match()` 成员函数, 正是此成员函数确定了 `platform_device` 和 `platform_driver` 之间如何匹配, 如代码清单 12.4 所示。

代码清单 12.4 `platform.bus.type` 的 `match()` 成员函数

```
1 static int platform_match(struct device *dev, struct device_driver *drv)  
2 {  
3     struct platform_device *pdev;  
4  
5     pdev = container_of(dev, struct platform_device, dev);  
6     return (strncmp(pdev->name, drv->name, BUS_ID_SIZE) == 0);  
7 }
```

从代码清单 12.4 的第 6 行可以看出, 匹配 `platform_device` 和 `platform_driver` 主要看两者的 `name` 字段是否相同。

对 `platform_device` 的定义通常在 BSP 的板文件中实现, 在板文件中, 将 `platform_device` 归纳为一个数组, 最终通过 `platform_add_devices()` 函数统一注册。`platform_add_devices()` 函数可以将平台设备添加到系统中, 这个函数的原型为:

```
int platform_add_devices(struct platform_device **devs, int num);
```

该函数的第一个参数为平台设备数组的指针, 第二个参数为平台设备的数量, 它内部调用了 `platform_device_register()` 函数用于注册单个的平台设备。

12.1.2 将 globalfifo 作为 platform 设备

现在我们将前面章节的 `globalfifo` 驱动挂接到 `platform` 总线上, 要完成两个工作。

- (1) 将 `globalfifo` 移植为 `platform` 驱动。
- (2) 在板文件中添加 `globalfifo` 这个 `platform` 设备。

为完成将 `globalfifo` 移植到 `platform` 驱动的工作, 需要在原始的 `globalfifo` 字符设备驱动中套一层 `platform_driver` 的外壳, 如代码清单 12.5。注意进行这一工作后, 并没有改变 `globalfifo` 是字符设备的本质, 只是将其挂接到了 `platform` 总线。

代码清单 12.5 为 `globalfifo` 添加 `platform.driver`

```
1 static int __devinit globalfifo_probe(struct platform_device *pdev)  
2 {  
3     int ret;  
4     dev_t devno = MKDEV(globalfifo_major, 0);  
5  
6     /* 申请设备号 */  
7     if (globalfifo_major)  
8         ret = register_chrdev_region(devno, 1, "globalfifo");  
9     else { /* 动态申请设备号 */  
10         ret = alloc_chrdev_region(&devno, 0, 1, "globalfifo");  
11         globalfifo_major = MAJOR(devno);  
12     }  
13     if (ret < 0)  
14         return ret;
```



```

15      /* 动态申请设备结构体的内存*/
16      globalfifo_devp = kmalloc(sizeof(struct globalfifo_dev), GFP_KERNEL);
17      if (!globalfifo_devp) { /*申请失败*/
18          ret = - ENOMEM;
19          goto fail_malloc;
20      }
21
22      memset(globalfifo_devp, 0, sizeof(struct globalfifo_dev));
23
24      globalfifo_setup_cdev(globalfifo_devp, 0);
25
26      init_MUTEX(&globalfifo_devp->sem); /*初始化信号量*/
27      init_waitqueue_head(&globalfifo_devp->r_wait); /*初始化读等待队列头*/
28      init_waitqueue_head(&globalfifo_devp->w_wait); /*初始化写等待队列头*/
29
30      return 0;
31
32 fail_malloc: unregister_chrdev_region(devno, 1);
33      return ret;
34 }
35
36 static int __devexit globalfifo_remove(struct platform_device *pdev)
37 {
38     cdev_del(&globalfifo_devp->cdev); /*注销 cdev*/
39     kfree(globalfifo_devp); /*释放设备结构体内存*/
40     unregister_chrdev_region(MKDEV(globalfifo_major, 0), 1); /*释放设备号*/
41     return 0;
42 }
43
44 static struct platform_driver globalfifo_device_driver = {
45     .probe      = globalfifo_probe,
46     .remove     = __devexit_p(globalfifo_remove),
47     .driver     = {
48         .name    = "globalfifo",
49         .owner   = THIS_MODULE,
50     }
51 };
52
53 static int __init globalfifo_init(void)
54 {
55     return platform_driver_register(&globalfifo_device_driver);
56 }
57
58 static void __exit globalfifo_exit(void)
59 {
60     platform_driver_unregister(&globalfifo_device_driver);
61 }
62
63 module_init(globalfifo_init);
64 module_exit(globalfifo_exit);

```

在代码清单 12.5 中，模块加载和卸载函数仅仅通过 `platform_driver_register()`、`platform_driver_unregister()` 函数进行 `platform_driver` 的注册与注销，而原先注册和注销字符设备的工作已经被移交到 `platform_driver` 的 `probe()` 和 `remove()` 成员函数中。

代码清单 12.5 未列出的部分与原始的 `globalfifo` 驱动相同，都是实现作为字符设备驱动核心的 `file_operations` 的成员函数。



为了完成在板文件中添加 globalfifo 这个 platform 设备的工作,需要在板文件(对于 LDD6410 而言,为 arch/arm/mach-s3c6410/ mach-ldd6410.c)中添加相应的代码,如代码清单 12.6 所示。

代码清单 12.6 globalfifo 对应的 platform.device

```
1 static struct platform_device globalfifo_device = {
2     .name      = "globalfifo",
3     .id        = -1,
4 };
```

对于 LDD6410 开发板而言,为了完成上述 globalfifo_device 这一 platform_device 的注册,只需要将其地址放入 arch/arm/mach-s3c6410/ mach-ldd6410.c 中定义的 ldd6410_devices 数组,如:

```
static struct platform_device *ldd6410_devices[]__initdata = {
+    & globalfifo_device,
#ifdef CONFIG_FB_S3C_V2
    &s3c_device_fb,
#endif
    &s3c_device_hsmmc0,
    ...
}
```

在加载 LDD6410 驱动后,在 sysfs 中会发现如下结点:

```
/sys/bus/platform/devices/globalfifo/
/sys/devices/platform/globalfifo/
```

留意一下代码清单 12.5 的第 48 行和代码清单 12.6 的第 2 行,platform_device 和 platform_driver 的 name 一致,这是两者得以匹配的前提。

12.1.3 platform 设备资源和数据

留意一下代码清单 12.1 中 platform_device 结构体定义的第 5~6 行,描述了 platform_device 的资源,资源本身由 resource 结构体描述,其定义如代码清单 12.7 所示。

代码清单 12.7 resource 结构体定义

```
1 struct resource {
2     resource_size_t start;
3     resource_size_t end;
4     const char *name;
5     unsigned long flags;
6     struct resource *parent, *sibling, *child;
7 };
```

我们通常关心 start、end 和 flags 这 3 个字段,分别标明资源的开始值、结束值和类型,flags 可以为 IORESOURCE_IO、IORESOURCE_MEM、IORESOURCE_IRQ、IORESOURCE_DMA 等。start、end 的含义会随着 flags 而变更,如当 flags 为 IORESOURCE_MEM 时,start、end 分别表示该 platform_device 占据的内存的开始地址和结束地址;当 flags 为 IORESOURCE_IRQ 时,start、end 分别表示该 platform_device 使用的中断号的开始值和结束值,如果只使用了 1 个中断号,开始和结束值相同。对于同种类型的资源而言,可以有多份,例如说某设备占据了 2 个内存区域,则可以定义 2 个 IORESOURCE_MEM 资源。

对 resource 的定义也通常在 BSP 的板文件中进行,而在具体的设备驱动中透过 platform_get_resource()这样的 API 来获取,此 API 的原型为:

```
struct resource *platform_get_resource(struct platform_device *, unsigned int, unsigned int);
```

例如在 LDD6410 开发板的板文件中为 DM9000 网卡定义了如下 resource:

```
static struct resource ldd6410_dm9000_resource[] = {
    [0] = {
        .start = 0x18000000,
        .end   = 0x18000000 + 3,
        .flags = IORESOURCE_MEM
    },
    [1] = {
        .start = 0x18000000 + 0x4,
        .end   = 0x18000000 + 0x7,
        .flags = IORESOURCE_MEM
    },
    [2] = {
        .start = IRQ_EINT(7),
        .end   = IRQ_EINT(7),
        .flags = IORESOURCE_IRQ | IORESOURCE_IRQ_HIGHLEVEL,
    }
};
```

在 DM9000 网卡的驱动中则是通过如下办法拿到这 3 份资源:

```
db->addr_res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
db->data_res = platform_get_resource(pdev, IORESOURCE_MEM, 1);
db->irq_res  = platform_get_resource(pdev, IORESOURCE_IRQ, 0);
```

对于 IRQ 而言, platform_get_resource() 还有一个进行了封装的变体 platform_get_irq(), 其原型为:

```
int platform_get_irq(struct platform_device *dev, unsigned int num);
```

它实际上调用了 “platform_get_resource(dev, IORESOURCE_IRQ, num);”。

设备除了可以在 BSP 中定义资源以外, 还可以附加一些数据信息, 因为对设备的硬件描述除了中断、内存、DMA 通道以外, 可能还会有一些配置信息, 而这些配置信息也依赖于板, 不适宜直接放置在设备驱动本身, 因此, platform 也提供了 platform_data 的支持。platform_data 的形式是自定义的, 如对于 DM9000 网卡而言, platform_data 为一个 dm9000_plat_data 结构体, 我们就可以将 MAC 地址、总线宽度、板上有无 EEPROM 信息等放入 platform_data:

```
static struct dm9000_plat_data ldd6410_dm9000_platdata = {
    .flags      = DM9000_PLATF_16BITONLY | DM9000_PLATF_NO_EEPROM,
    .dev_addr   = { 0x0, 0x16, 0xd4, 0x9f, 0xed, 0xa4 },
};

static struct platform_device ldd6410_dm9000 = {
    .name=      "dm9000",
    .id=        0,
    .num_resources= ARRAY_SIZE(ldd6410_dm9000_resource),
    .resource   =ldd6410_dm9000_resource,
    .dev       = {
        .platform_data = &ldd6410_dm9000_platdata,
    }
};
```

而在 DM9000 网卡的驱动中, 通过如下方式就拿到了 platform_data:

```
struct dm9000_plat_data *pdata = pdev->dev.platform_data;
```

其中, pdev 为 platform_device 的指针。

由以上分析可知, 设备驱动中引入 platform 的概念至少有如下两大好处。



(1) 使得设备被挂接在一个总线上, 因此, 符合 Linux 2.6 的设备模型。其结果是, 配套的 sysfs 结点、设备电源管理都成为可能。

(2) 隔离 BSP 和驱动。在 BSP 中定义 platform 设备和设备使用的资源、设备的具体配置信息, 而在驱动中, 只需要通过通用 API 去获取资源和数据, 做到了板相关代码和驱动代码的分离, 使得驱动具有更好的可扩展性和跨平台性。

12.2 设备驱动的分层思想

12.2.1 设备驱动核心层和例化

在面向对象的程序设计中, 可以为某一类相似的事物定义一个基类, 而具体的事物可以继承这个基类中的函数。如果对于继承的这个事物而言, 其某成员函数的实现与基类一致, 那它就可以直接继承基类的函数; 相反, 它可以重载之。这种面向对象的设计思想极大地提高了代码的可重用能力, 是对现实世界事物间关系的一种良好呈现。

Linux 内核完全由 C 语言和汇编语言写成, 但是却频繁用到了面向对象的设计思想。在设备驱动方面, 往往为同类的设备设计了一个框架, 而框架中的核心层则实现了该设备通用的一些功能。同样的, 如果具体的设备不想使用核心层的函数, 它可以重载之。举个例子:

```
return_type core_funca (xxx_device * bottom_dev, param1_type param1, param1_type param2)
{
    if (bottom_dev->funca)
        return bottom_dev->funca (param1, param2);
    /* 核心层通用的 funca 代码 */
    ...
}
```

上述 core_funca 的实现中, 会检查底层设备是否重载了 funca(), 如果重载了, 就调用底层的代码, 否则, 直接使用通用层的。这样做的好处是, 核心层的代码可以处理绝大多数该类设备的 funca() 对应的功能, 只有少数特殊设备需要重新实现 funca()。

再看一个例子:

```
return_type core_funca (xxx_device * bottom_dev, param1_type param1, param1_type param2)
{
    /* 通用的步骤代码 A */
    typea_dev_commonA ();
    ...

    /* 底层操作 ops1 */
    bottom_dev->funca_ops1 ();

    /* 通用的步骤代码 B */
    typea_dev_commonB ();
    ...

    /* 底层操作 ops2 */
    bottom_dev->funca_ops2 ();
}
```

```

/*通用的步骤代码 C */
typea_dev_commonB();
...

/** 底层操作 ops3 */
bottom_dev->funca_ops3();
}

```

上述代码假定为了实现 funca(), 对于同类设备而言, 操作流程一致, 都要经过“通用代码 A、底层 ops1、通用代码 B、底层 ops2、通用代码 C、底层 ops3”这几步, 分层设计明显带来的好处是, 对于通用代码 A、B、C, 具体的底层驱动不需要再实现, 而仅仅只关心其底层的操作 ops1、ops2、ops3。

图 12.1 明确反映了设备驱动的核心层与具体设备驱动的关系, 实际上, 这种分层可能只有两层 (图 12.1 的 a), 也可能是多层的 (图 12.1 的 b)。

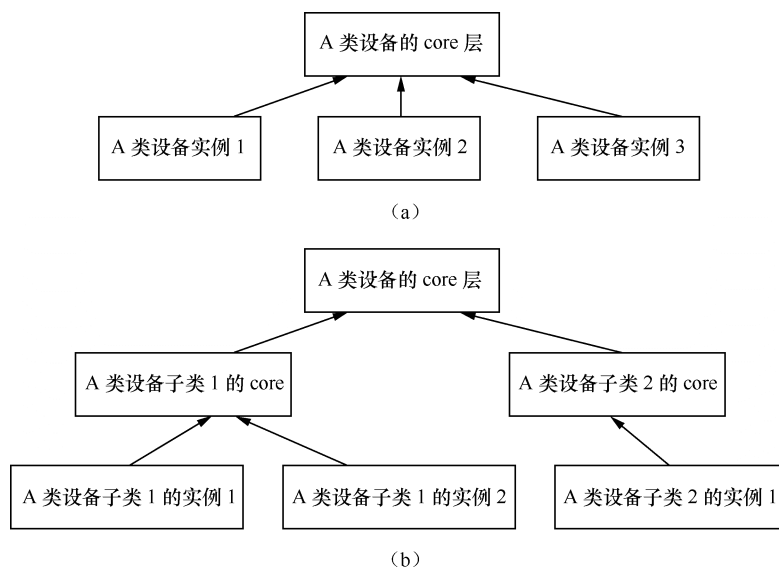


图 12.1 Linux 设备驱动的分层

这样的分层化设计在 Linux 的 input、RTC、MTD、I²C、SPI、TTY、USB 等诸多设备驱动类型中屡见不鲜。下面的两小节以 input 和 RTC 为例先行进行一番讲解, 当然, 后续的章节会对几个大的设备类型对应驱动的层次进行更详细的分析。

12.2.2 输入设备驱动

输入设备 (如按键、键盘、触摸屏、鼠标等) 是典型的字符设备, 其一般的工作机理是底层在按键、触摸等动作发送时产生一个中断 (或驱动通过 timer 定时查询), 然后 CPU 通过 SPI、I²C 或外部存储器总线读取键值、坐标等数据, 放入一个缓冲区, 字符设备驱动管理该缓冲区, 而驱动的 read() 接口让用户可以读取键值、坐标等数据。

显然, 在这些工作中, 只是中断、读键值/坐标值是设备相关的, 而输入事件的缓冲区管理以及字符设备驱动的 file_operations 接口则对输入设备是通用的。基于此, 内核设计了输入子系统,



由核心层处理公共的工作。Linux 内核输入子系统的框架如图 12.2 所示。

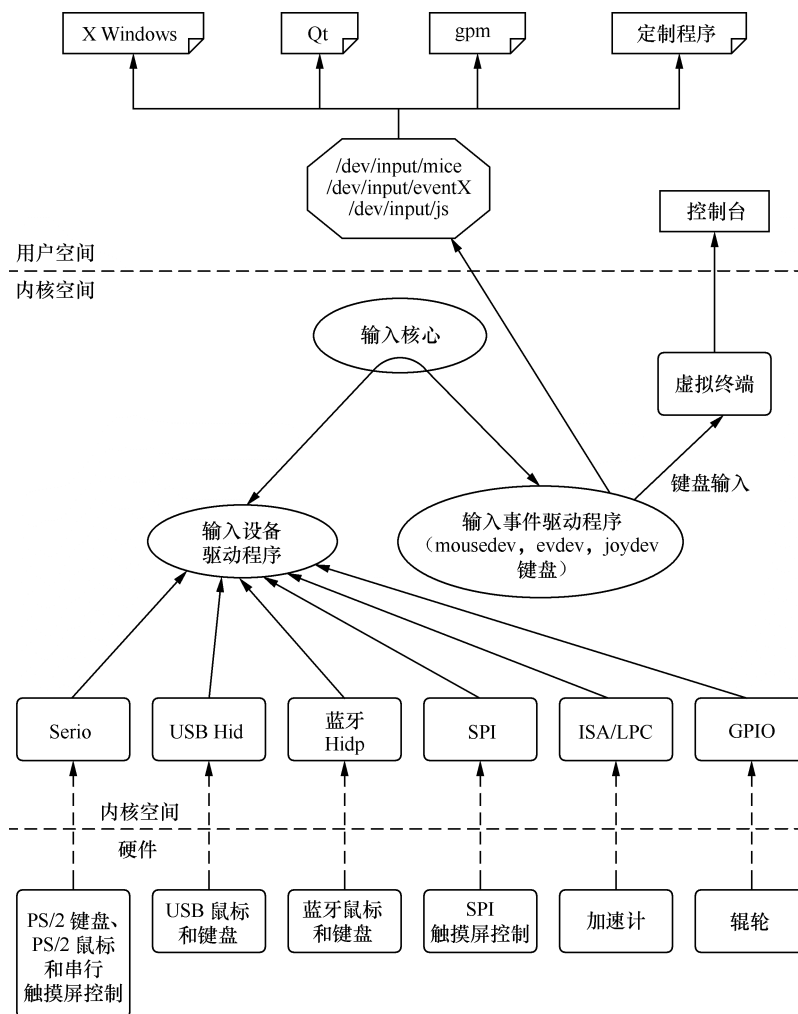


图 12.2 Linux 输入设备驱动的分层

输入核心提供了底层输入设备驱动程序所需的 API，如分配/释放一个输入设备：

```
struct input_dev *input_allocate_device(void);
void input_free_device(struct input_dev *dev);
```

`input_allocate_device()` 返回的是 1 个 `input_dev` 的结构体，此结构体用于表征 1 个输入设备。

注册/注销输入设备用的接口如下：

```
int __must_check input_register_device(struct input_dev *);
void input_unregister_device(struct input_dev *);
```

报告输入事件用的接口如下：

```
/* 报告指定 type、code 的输入事件 */
void input_event(struct input_dev *dev, unsigned int type, unsigned int code, int value);
/* 报告键值 */
void input_report_key(struct input_dev *dev, unsigned int code, int value);
/* 报告相对坐标 */
void input_report_rel(struct input_dev *dev, unsigned int code, int value);
```

```

/* 报告绝对坐标 */
void input_report_abs(struct input_dev *dev, unsigned int code, int value);
/* 报告同步事件 */
void input_sync(struct input_dev *dev);

```

而所有的输入事件，内核都用统一的数据结构来描述，这个数据结构是 `input_event`，形如代码清单 12.8。

代码清单 12.8 `input_event` 结构体

```

1 struct input_event {
2     struct timeval time;
3     __u16 type;
4     __u16 code;
5     __s32 value;
6 };

```

`drivers/input/keyboard/gpio_keys.c` 基于 `input` 架构实现了一个通用的 GPIO 按键驱动。该驱动基于 `platform_driver` 架构，名为“`gpio-keys`”。它将硬件相关的信息（如使用的 GPIO 号，按下和抬起时的电平）屏蔽在板文件 `platform_device` 的 `platform_data` 中，因此该驱动可应用于各个处理器，具有良好的跨平台性。代码清单 12.9 列出了该驱动的 `probe()` 函数。

代码清单 12.9 GPIO 按键驱动的 `probe()` 函数

```

1 static int __devinit gpio_keys_probe(struct platform_device *pdev)
2 {
3     struct gpio_keys_platform_data *pdata = pdev->dev.platform_data;
4     struct gpio_keys_drvdata *ddata;
5     struct input_dev *input;
6     int i, error;
7     int wakeup = 0;
8
9     ddata = kzalloc(sizeof(struct gpio_keys_drvdata) +
10                    pdata->nbuttons * sizeof(struct gpio_button_data),
11                    GFP_KERNEL);
12     input = input_allocate_device();
13     if (!ddata || !input) {
14         error = -ENOMEM;
15         goto fail1;
16     }
17
18     platform_set_drvdata(pdev, ddata);
19
20     input->name = pdev->name;
21     input->phys = "gpio-keys/input0";
22     input->dev.parent = &pdev->dev;
23
24     input->id.bustype = BUS_HOST;
25     input->id.vendor = 0x0001;
26     input->id.product = 0x0001;
27     input->id.version = 0x0100;
28
29     ddata->input = input;
30
31     for (i = 0; i < pdata->nbuttons; i++) {
32         struct gpio_keys_button *button = &pdata->buttons[i];

```



```
33     struct gpio_button_data *bdata = &ddata->data[i];
34     int irq;
35     unsigned int type = button->type ?: EV_KEY;
36
37     bdata->input = input;
38     bdata->button = button;
39     setup_timer(&bdata->timer,
40               gpio_check_button, (unsigned long)bdata);
41
42     ...
43     error = request_irq(irq, gpio_keys_isr,
44                       IRQF_SAMPLE_RANDOM | IRQF_TRIGGER_RISING |
45                       IRQF_TRIGGER_FALLING,
46                       button->desc ? button->desc : "gpio_keys",
47                       bdata);
48     if (error) {
49         ...
50     }
51
52     if (button->wakeup)
53         wakeup = 1;
54
55     input_set_capability(input, type, button->code);
56 }
57
58 error = input_register_device(input);
59 if (error) {
60     pr_err("gpio-keys: Unable to register input device, "
61           "error: %d\n", error);
62     goto fail2;
63 }
64
65 device_init_wakeup(&pdev->dev, wakeup);
66
67 return 0;
68 ...
69 }
```

上述代码的第 12 行分配了 1 个输入设备, 第 20~27 行初始化了该 `input_dev` 的一些属性, 第 58 行注册了这个输入设备。第 31~56 行则申请了此 GPIO 按键设备需要的中断号, 并初始化了 `timer`。第 55 行设置此输入设备可告知的事情。

在注册输入设备后, 底层输入设备驱动的核心工作只剩下在按键、触摸等人为动作发生的时候, 报告事件。代码清单 12.10 列出了 GPIO 按键中断发生时的事件报告代码。

代码清单 12.10 GPIO 按键中断发生时的事件报告

```
1 static void gpio_keys_report_event(struct gpio_button_data *bdata)
2 {
3     struct gpio_keys_button *button = bdata->button;
4     struct input_dev *input = bdata->input;
5     unsigned int type = button->type ?: EV_KEY;
6     int state = (gpio_get_value(button->gpio) ? 1 : 0) ^ button->active_low;
7 }
```



```

8  input_event(input, type, button->code, !!state);
9  input_sync(input);
10 }
11
12 static irqreturn_t gpio_keys_isr(int irq, void *dev_id)
13 {
14     struct gpio_button_data *bdata = dev_id;
15     struct gpio_keys_button *button = bdata->button;
16
17     BUG_ON(irq != gpio_to_irq(button->gpio));
18
19     if (button->debounce_interval)
20         mod_timer(&bdata->timer,
21                 jiffies + msecs_to_jiffies(button->debounce_interval));
22     else
23         gpio_keys_report_event(bdata);
24
25     return IRQ_HANDLED;
26 }

```

第 8 行是报告键值，而第 9 行是 1 个同步事件，暗示前面报告的消息属于 1 个消息组。例如，用户在报告完 X 坐标后，又报告 Y 坐标，之后报告 1 个同步事件，应用程序即可知道前面报告的 X、Y 这两个事件属于 1 组，它会将两者联合起来形成 1 个 (X,Y) 的坐标。

代码清单 12.8 第 2 行获取 platform_data，而 platform_data 实际上是定义 GPIO 按键硬件信息的数组，第 31 行的 for 循环工具这些信息申请 GPIO 并初始化中断，对于 LDD6140 电路板而言，这些信息如代码清单 12.11。

代码清单 12.11 LDD6410 开发板 GPIO 按键的 platform.data

```

1  static struct gpio_keys_button ldd6410_buttons[] = {
2      {
3          .gpio      = S3C64XX_GPN(0),
4          .code       = KEY_DOWN,
5          .desc       = "Down",
6          .active_low = 1,
7      },
8      {
9          .gpio      = S3C64XX_GPN(1),
10         .code       = KEY_ENTER,
11         .desc       = "Enter",
12         .active_low = 1,
13         .wakeup     = 1,
14     },
15     {
16         .gpio      = S3C64XX_GPN(2),
17         .code       = KEY_HOME,
18         .desc       = "Home",
19         .active_low = 1,
20     },
21     {
22         .gpio      = S3C64XX_GPN(3),

```



```
23     .code      = KEY_POWER,
24     .desc      = "Power",
25     .active_low = 1,
26     .wakeup     = 1,
27 },
28 {
29     .gpio      = S3C64XX_GPN(4),
30     .code      = KEY_TAB,
31     .desc      = "Tab",
32     .active_low = 1,
33 },
34 {
35     .gpio      = S3C64XX_GPN(5),
36     .code      = KEY_MENU,
37     .desc      = "Menu",
38     .active_low = 1,
39 },
40 };
41
42 static struct gpio_keys_platform_data ldd6410_button_data = {
43     .buttons = ldd6410_buttons,
44     .nbuttons = ARRAY_SIZE(ldd6410_buttons),
45 };
46
47 static struct platform_device ldd6410_device_button = {
48     .name      = "gpio-keys",
49     .id        = -1,
50     .dev       = {
51         .platform_data = &ldd6410_button_data,
52     }
53 };
```

12.2.3 RTC 设备驱动

RTC (实时钟) 借助电池供电, 在系统掉电的情况下时间依然可以正常走动。它通常还具有产生周期性中断以及产生闹钟 (alarm) 中断的能力, 是一种典型的字符设备。作为一种字符设备驱动, RTC 需要有 `file_operations` 中接口函数的实现, 如 `open()`、`release()`、`read()`、`poll()`、`ioctl()` 等, 而典型的 `IOCTL` 包括 `RTC_SET_TIME`、`RTC_ALM_READ`、`RTC_ALM_SET`、`RTC_IRQP_SET`、`RTC_IRQP_READ` 等, 这些对于所有的 RTC 是通用的, 只有底层的具体实现是设备相关的。

因此, `drivers/rtc/rtc-dev.c` 实现了 RTC 驱动通用的字符设备驱动层, 它实现了 `file_operations` 的成员函数以及一些关于 RTC 的通用的控制代码, 并向底层导出 `rtc_device_register()`、`rtc_device_unregister()` 用于注册和注销 RTC; 导出 `rtc_class_ops` 结构体用于描述底层的 RTC 硬件操作。这一 RTC 通用层实现的结果是, 底层的 RTC 驱动不再需要关心 RTC 作为字符设备驱动的具体实现, 也无需关心一些通用的 RTC 控制逻辑, 图 12.3 表明了这种关系。

`drivers/rtc/rtc-s3c.c` 实现了 S3C6410 的 RTC 驱动, 其注册 RTC 以及绑定的 `rtc_class_ops` 的代码如代码清单 12.12 所示。

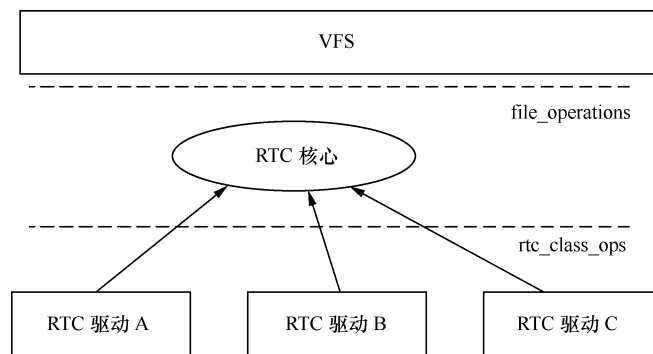


图 12.3 Linux RTC 设备驱动的分层

代码清单 12.12 S3C6410 RTC 驱动的 rtc.class.ops 实例与 RTC 注册

```

1 static const struct rtc_class_ops s3c_rtcops = {
2     .open      = s3c_rtc_open,
3     .release   = s3c_rtc_release,
4     .ioctl     = s3c_rtc_ioctl,
5     .read_time = s3c_rtc_gettime,
6     .set_time  = s3c_rtc_settime,
7     .read_alarm = s3c_rtc_getalarm,
8     .set_alarm = s3c_rtc_setalarm,
9     .irq_set_freq = s3c_rtc_setfreq,
10    .irq_set_state = s3c_rtc_setpie,
11    .proc       = s3c_rtc_proc,
12 };
13
14 static int s3c_rtc_probe(struct platform_device *pdev)
15 {
16     ...
17     rtc = rtc_device_register("s3c", &pdev->dev, &s3c_rtcops,
18                               THIS_MODULE);
19     ...
20 }

```

12.3 主机驱动与外设驱动分离思想

12.3.1 主机、外设驱动分离的意义

在 Linux 设备驱动框架的设计中，除了有分层设计实现以外，还有分隔的思想。举一个简单的例子，假设我们要通过 SPI 总线访问某外设，在这个访问过程中，要通过操作 CPU XXX 上的 SPI 控制器的寄存器来达到访问 SPI 外设 YYY 的目的，最简单的方法是：

```

return_type xxx_write_spi_yyy(...)
{
    xxx_write_spi_host_ctrl_reg(ctrl);
    xxx_write_spi_host_data_reg(buf);
}

```



```
while(!(xxx_spi_host_status_reg() & SPI_DATA_TRANSFER_DONE));  
...  
}
```

如果按照这种方式来设计驱动, 结果是对于任何一个 SPI 外设来讲, 它的驱动代码都是 CPU 相关的。也就是说, 当然用在 CPU XXX 上的时候, 它访问 XXX 的 SPI 主机控制寄存器, 当用在 XXX1 的时候, 它访问 XXX1 的 SPI 主机控制寄存器:

```
return_type xxx1_write_spi_yyy(...)  
{  
    xxx1_write_spi_host_ctrl_reg(ctrl);  
    xxx1_write_spi_host_data_reg(buf);  
  
    while(!(xxx1_spi_host_status_reg() & SPI_DATA_TRANSFER_DONE));  
    ...  
}
```

这显然是不能接受的, 因为这意味着外设 YYY 用在不同的 CPU XXX 和 XXX1 上的时候需要不同的驱动。那么, 我们可以用如图 12.4 所示的思想对主机控制器驱动和外设驱动进行分离。这样的结果是, 外设 a、b、c 的驱动与主机控制器 A、B、C 的驱动不相关, 主机控制器驱动不关心外设, 而外设驱动也不关心主机, 外设只是访问核心层的通用的 API 进行数据传输, 主机和外设之间可以进行任意的组合。

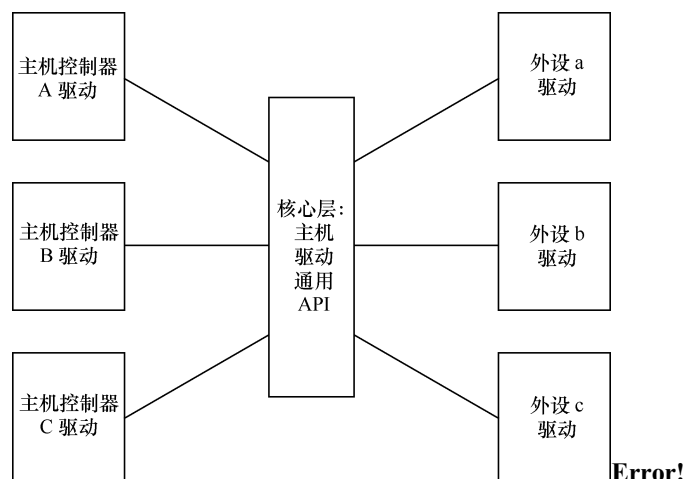


图 12.4 Linux 设备驱动的主机、外设驱动分离

如果我们不进行如图 12.4 所示的主机和外设分离, 外设 a、b、c 和主机 A、B、C 进行组合的时候, 需要 9 个不同的驱动。设想一共有 m 个主机控制器, n 个外设, 分离的结果是需要 $m + n$ 个驱动, 不分离则需要 $m * n$ 个驱动。

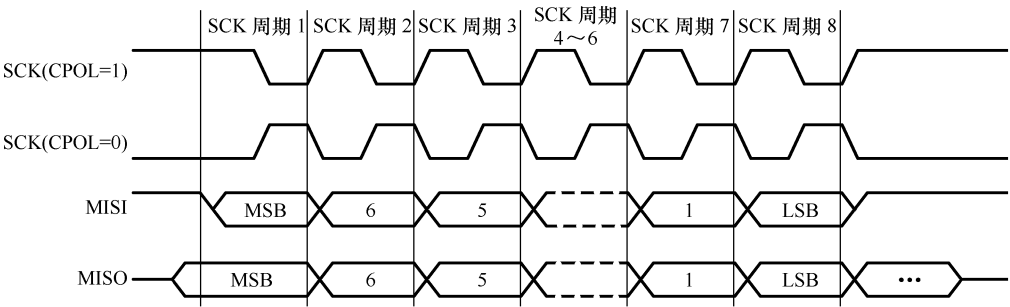
Linux SPI、I²C、USB、ASoC (ALSA SoC) 等子系统都典型地利用了这种分离的设计思想, 在本章我们先以简单一些的 SPI 为例, 而 I²C、USB、ASoC 等则在后续章节会进行详细介绍。

12.3.2 Linux SPI 主机和设备驱动

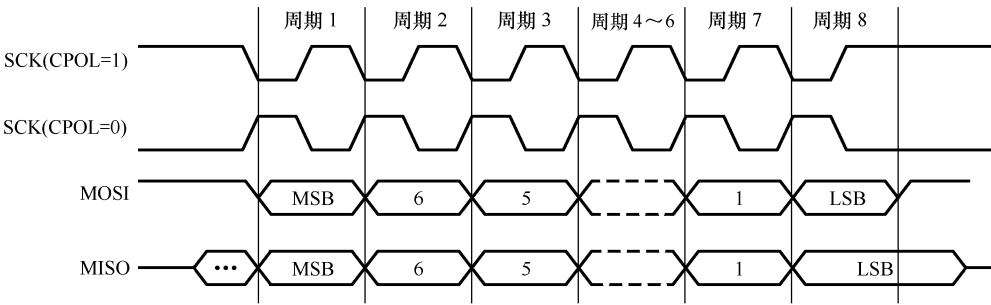
SPI (同步外设接口) 是由摩托罗拉公司开发的全双工同步串行总线, 其接口由 MISO (串行

数据输入)、MOSI (串行数据输出)、SCK (串行移位时钟)、SS (从使能信号) 4 种信号构成, SS 决定了惟一的与主设备通信的从设备, 主设备通过产生移位时钟来发起通信。通信时, 数据由 MOSI 输出, MISO 输入, 数据在时钟的上升或下降沿由 MOSI 输出, 在紧接着的下降或上升沿由 MISO 读入, 这样经过 8/16 次时钟的改变, 完成 8/16 位数据的传输。

SPI 模块为了和外设进行数据交换, 根据外设工作要求, 其输出串行同步时钟极性 (CPOL) 和相位 (CPHA) 可以进行配置。如果 CPOL=0, 串行同步时钟的空闲状态为低电平; 如果 CPOL=1, 串行同步时钟的空闲状态为高电平。如果 CPHA=0, 在串行同步时钟的第一个跳变沿 (上升或下降) 数据被采样; 如果 CPHA=1, 在串行同步时钟的第二个跳变沿 (上升或下降) 数据被采样。SPI 接口时序如图 12.5 所示。



CPHA=0 时 SPI 总线数据传输时序



CPHA=1 时 SPI 总线数据传输时序

图 12.5 SPI 总线时序

在 Linux 中, 用代码清单 12.13 的 spi_master 结构体来描述一个 SPI 主机控制器驱动, 其主要成员是主机控制器的序号 (系统中可能存在多个 SPI 主机控制器)、片选数量、SPI 模式和时钟设置用到的函数、数据传输用到的函数等。

代码清单 12.13 spi.master 结构体

```
1 struct spi_master {
2     struct device dev;
3     s16 bus_num;
4     u16 num_chipselect;
5 }
```



```
6    /* 设置模式和时钟 */
7    int                (*setup)(struct spi_device *spi);
8
9    /* 双向数据传输 */
10   int                (*transfer)(struct spi_device *spi,
11                                   struct spi_message *mesg);
12
13   void                (*cleanup)(struct spi_device *spi);
14 };
```

分配、注册和注销 SPI 主机的 API 由 SPI 核心提供:

```
struct spi_master * spi_alloc_master(struct device *host, unsigned size);
int spi_register_master(struct spi_master *master);
void spi_unregister_master(struct spi_master *master);
```

在 Linux 中, 用代码清单 12.14 的 `spi_driver` 结构体来描述一个 SPI 外设驱动, 可以认为是 `spi_master` 的 client 驱动。

代码清单 12.14 `spi_driver` 结构体

```
1 struct spi_driver {
2     int                (*probe)(struct spi_device *spi);
3     int                (*remove)(struct spi_device *spi);
4     void               (*shutdown)(struct spi_device *spi);
5     int                (*suspend)(struct spi_device *spi, pm_message_t mesg);
6     int                (*resume)(struct spi_device *spi);
7     struct device_driver driver;
8 };
```

可以看出, `spi_driver` 结构体和 `platform_driver` 结构体有极大的相似性, 都有 `probe()`、`remove()`、`suspend()`、`resume()` 这样的接口。是的, 这几乎是一切 client 驱动的习惯模板。

在 SPI 外设驱动中, 当透过 SPI 总线进行数据传输的时候, 使用了一套与 CPU 无关的统一的接口。这套接口的第 1 个关键数据结构就是 `spi_transfer`, 它用于描述 SPI 传输, 如代码清单 12.15 所示。

代码清单 12.15 `spi_transfer` 结构体

```
1 struct spi_transfer {
2     const void        *tx_buf;
3     void              *rx_buf;
4     unsigned          len;
5
6     dma_addr_t        tx_dma;
7     dma_addr_t        rx_dma;
8
9     unsigned          cs_change:1;
10    u8                 bits_per_word;
11    u16                 delay_usecs;
12    u32                 speed_hz;
13
14    struct list_head transfer_list;
15 };
```

而一次完整的 SPI 传输流程可能不只包含一次 `spi_transfer`, 它可能包含一个或多个 `spi_transfer`, 这些 `spi_transfer` 最终通过 `spi_message` 组织在一起, 其定义如代码清单 12.16 所示。

代码清单 12.16 spi_message 结构体

```

1 struct spi_message {
2     struct list_head    transfers;
3
4     struct spi_device    *spi;
5
6     unsigned            is_dma_mapped:1;
7
8     /* 完成被一个 callback 报告 */
9     void                (*complete)(void *context);
10    void                *context;
11    unsigned            actual_length;
12    int                 status;
13
14    struct list_head    queue;
15    void                *state;
16 };

```

通过 `spi_message_init()` 可以初始化 `spi_message`，而将 `spi_transfer` 添加到 `spi_message` 队列的方法则是：

```
void spi_message_add_tail(struct spi_transfer *t, struct spi_message *m);
```

发起一次 `spi_message` 的传输有同步和异步两种方式，使用同步 API 时，会阻塞等待这个消息被处理完。同步操作时使用的 API 是：

```
int spi_sync(struct spi_device *spi, struct spi_message *message);
```

使用异步 API 时，不会阻塞等待这个消息被处理完，但是可以在 `spi_message` 的 `complete` 字段挂接一个回调函数，当消息被处理完成后，该函数会被调用。异步操作时使用的 API 是：

```
int spi_async(struct spi_device *spi, struct spi_message *message);
```

代码清单 12.17 是非常典型的初始化 `spi_transfer`、`spi_message` 并进行 SPI 数据传输的例子，同时它们也是 SPI 核心层的两个通用 API，在 SPI 外设驱动中可以直接调用它们进行写和读操作。

代码清单 12.17 SPI 传输实例 `spi.write()`、`spi.read()` API

```

1 static inline int
2 spi_write(struct spi_device *spi, const u8 *buf, size_t len)
3 {
4     struct spi_transfer    t = {
5         .tx_buf            = buf,
6         .len               = len,
7     };
8     struct spi_message    m;
9
10    spi_message_init(&m);
11    spi_message_add_tail(&t, &m);
12    return spi_sync(spi, &m);
13 }
14
15 static inline int
16 spi_read(struct spi_device *spi, u8 *buf, size_t len)
17 {
18     struct spi_transfer    t = {
19         .rx_buf            = buf,
20         .len               = len,

```



```
21     };
22     struct spi_message    m;
23
24     spi_message_init(&m);
25     spi_message_add_tail(&t, &m);
26     return spi_sync(spi, &m);
27 }
```

LDD6410 开发板所使用的 S3C6410 的 SPI 主机控制器驱动位于 `drivers/spi/spi_s3c.h` 和 `drivers/spi/spi_s3c.c` 这两个文件, 其主体是实现了 `spi_master` 的 `setup()`、`transfer()` 等成员函数。

SPI 外设驱动遍布于内核的 `drivers`、`sound` 的各个子目录之下, SPI 只是一种总线, `spi_driver` 的作用只是将 SPI 外设挂接在该总线上, 因此在 `spi_driver` 的 `probe()` 成员函数中, 将注册 SPI 外设本身所属设备驱动的类型。

和 `platform_driver` 对应着一个 `platform_device` 一样, `spi_driver` 也对应着一个 `spi_device`; `platform_device` 需要在 BSP 的板文件中添加板信息数据, 而 `spi_device` 也同样需要。 `spi_device` 的板信息用 `spi_board_info` 结构体描述, 该结构体记录 SPI 外设使用的主机控制器序号、片选序号、数据比特率、SPI 传输模式 (即 CPOL、CPHA) 等。如诺基亚 770 上两个 SPI 设备的板信息数据如代码清单 12.18, 位于板文件 `arch/arm/mach-omap1/board-nokia770.c`。

代码清单 12.18 诺基亚 770 板文件中的 `spi_board_info`

```
1 static struct spi_board_info nokia770_spi_board_info[] __initdata = {
2     [0] = {
3         .modalias      = "lcd_mipid",
4         .bus_num        = 2, /* 用到的 SPI 主机控制器序号 */
5         .chip_select    = 3, /* 使用哪一号片选 */
6         .max_speed_hz   = 12000000, /* SPI 数据传输比特率 */
7         .platform_data  = &nokia770_mipid_platform_data,
8     },
9     [1] = {
10        .modalias      = "ads7846",
11        .bus_num        = 2,
12        .chip_select    = 0,
13        .max_speed_hz   = 2500000,
14        .irq            = OMAP_GPIO_IRQ(15),
15        .platform_data  = &nokia770_ads7846_platform_data,
16    },
17 };
```

在 Linux 启动过程中, 在机器的 `init_machine()` 函数中, 会通过如下语句注册这些 `spi_board_info`:

```
spi_register_board_info(nokia770_spi_board_info,
                        ARRAY_SIZE(nokia770_spi_board_info));
```

这一点和启动时通过 `platform_add_devices()` 添加 `platform_device` 非常相似。

12.4 设备驱动中的电源管理

一个真实生活中的设备驱动除了要处理设备的基本功能以外, 还需要处理电源管理, 主要提供挂起和恢复用的 `suspend()`、`resume()` 两个函数, 对于 `platform_driver` 而言, 该结构体已经包含

了这两个成员函数，包含 `suspend()`、`resume()` 入口的 `platform_driver` 一般形如代码清单 12.19。

代码清单 12.19 包含 `suspend/resume` 的 `platform_driver`

```

1  #ifdef CONFIG_PM
2  static int xxx_suspend(struct platform_device *pdev, pm_message_t state)
3  {
4  ...
5  return 0;
6  }
7
8  static int xxx_resume(struct platform_device *pdev)
9  {
10 ...
11 return 0;
12 }
13 #else
14 #define xxx_suspend NULL
15 #define xxx_resume  NULL
16 #endif
17
18 static struct platform_driver xxx_driver = {
19     .probe      = xxx_probe,
20     .remove     = xxx_remove,
21     .suspend    = xxx_suspend,
22     .resume     = xxx_resume,
23     .driver     = {
24         .name    = "xxx",
25         .owner   = THIS_MODULE,
26     },
27 };

```

上述代码清单中，对于 `suspend()`、`resume()` 进行了 `CONFIG_PM` 宏的检查，也就是说内核配置了电源管理的情况下，才定义 `suspend()`、`resume()` 的实体，否则将它们定义为 `NULL`。

通常而言，在 `suspend()` 函数里面会停止设备，并关闭给它提供的时钟，所以在 `suspend()` 函数里面经常看见这样的语句：

```
clk_disable(xxx->clk);
```

而在 `resume()` 函数中，进行相反的操作：

```
clk_enable(xxx->clk);
```

`clk_disable()`、`clk_enable()` 的具体实现直接依赖于 SoC 的类型，实际上，在 BSP 内为 SoC 内的各个 PLL、分频器和时钟 `gate` 建立了一颗树，并提供了一组操作时钟的通用 API。因此，在具体的设备驱动中，最好不要直接去修改寄存器来操作时钟，而应该用如下 API：

```

/* 获得、释放时钟 */
struct clk *clk_get(struct device *dev, const char *id);
void clk_put(struct clk *clk);

/* 使能、禁止时钟 */
int clk_enable(struct clk *clk);
void clk_disable(struct clk *clk);

/* 获得、试探和设置频率 */
unsigned long clk_get_rate(struct clk *clk);

```



```
long clk_round_rate(struct clk *clk, unsigned long rate);
int clk_set_rate(struct clk *clk, unsigned long rate);
```

```
/* 设置、获得父时钟 */
```

```
int clk_set_parent(struct clk *clk, struct clk *parent);
struct clk *clk_get_parent(struct clk *clk);
```

从代码清单 12.2 中 platform_driver 结构体的定义可知, 它除了包含 suspend()和 resume()入口以外, 还包含如下两个入口:

```
int (*suspend_late)(struct platform_device *, pm_message_t state);
int (*resume_early)(struct platform_device *);
```

suspend_late()与 suspend()的区别在于, suspend_late()工作于中断都被禁止的情况下, 而且仅有一个 CPU 是活跃的。相似的, resume_early()也工作于中断都被禁止的情况下。绝大多数情况下, 设备驱动不提供 suspend_late()和 resume_early()入口。

12.5 misc 设备驱动

Linux 包含了许多设备驱动类型, 而不管分类有多细, 总会有些漏网的, 这就是我们经常说到的“其他的”、“等等”。在 Linux 里面, 把无法归类的五花八门的设备定义为混杂设备(用 miscdevice 结构体描述)。Linux 内核所提供的 miscdevice 有很强的包容性, 如 NVRAM、看门狗、DS1286 等实时钟、字符 LCD、AMD 768 随机数发生器等, 体现了大杂烩的本意。

miscdevice 共享一个主设备号 MISC_MAJOR (即 10), 但次设备号不同。所有的 miscdevice 设备形成一个链表, 对设备访问时内核根据次设备号查找对应的 miscdevice 设备, 然后调用其 file_operations 结构体中注册的文件操作接口进行操作。

在内核中, 用 struct miscdevice 结构体表征 miscdevice 设备, 这个结构体的定义如代码清单 12.20 所示。

代码清单 12.20 miscdevice 结构体

```
1 struct miscdevice {
2     int minor;
3     const char *name;
4     const struct file_operations *fops;
5     struct list_head list;
6     struct device *parent;
7     struct device *this_device;
8 };
```

miscdevice 在本质上仍然属于字符设备, 只是被增加了一层封装而已, 因此其驱动的主体工作还是 file_operations 的成员函数。代码清单 12.21 则给出了源代码 drivers/char/nvram.c 所实现 NVRAM 驱动的 miscdevice 和 file_operations 实例。

代码清单 12.21 NVRAM 设备结构体

```
1 static const struct file_operations nvram_fops = {
2     .owner      = THIS_MODULE,
3     .llseek     = nvram_llseek,
4     .read       = nvram_read,
```

```

5     .write      = nvram_write,
6     .ioctl      = nvram_ioctl,
7     .open       = nvram_open,
8     .release    = nvram_release,
9 };
10
11 static struct miscdevice nvram_dev = {
12     NVRAM_MINOR,
13     "nvram",
14     &nvram_fops
15 };

```

对 `miscdevice` 的注册和注销分别通过如下两个 API 完成：

```

int misc_register(struct miscdevice * misc);
int misc_deregister(struct miscdevice *misc);

```

查看 `drivers/char/nvram.c` 的模块加载和卸载函数可知，其在加载的时候调用了“`misc_register(&nvram_dev);`”，而在模块卸载时调用了“`misc_deregister(&nvram_dev);`”。

12.6 基于 sysfs 的设备驱动

一些设备驱动以 `sysfs` 结点的形式存在，其本身没有对应的 `/dev` 结点；一些设备驱动虽然有对应的 `/dev` 结点，也依赖于 `sysfs` 结点进行一些工作。

Linux 专门提供了一种类型的设备驱动，以结构体 `sysdev_driver` 进行描述，该结构体的定义如代码清单 12.22 所示。

代码清单 12.22 `sysdev_driver`

```

1 struct sysdev_driver {
2     struct list_head    entry;
3     int      (*add) (struct sys_device *);
4     int      (*remove) (struct sys_device *);
5     int      (*shutdown) (struct sys_device *);
6     int      (*suspend) (struct sys_device *, pm_message_t state);
7     int      (*resume) (struct sys_device *);
8 };

```

注册和注销此类驱动的 API 为：

```

int sysdev_driver_register(struct sysdev_class *, struct sysdev_driver *);
void sysdev_driver_unregister(struct sysdev_class *, struct sysdev_driver *);

```

而此类驱动中通常会通过如下两个 API 来创建和移除 `sysfs` 的结点：

```

int sysdev_create_file(struct sys_device *, struct sysdev_attribute *);
void sysdev_remove_file(struct sys_device *, struct sysdev_attribute *);

```

而 `sysdev_create_file()` 最终调用的是 `sysfs_create_file()`，`sysfs_create_file()` 的第一个参数为 `kobject` 的指针，第二个参数是一个 `attribute` 结构体，每个 `attribute` 对应着 `sysfs` 中的一个文件，而读写一个 `attribute` 对应的文件通常需要 `show()` 和 `store()` 这两个函数，形如：

```

static ssize_t xxx_show(struct kobject * kobj, struct attribute * attr, char * buffer);
static ssize_t xxx_store(struct kobject * kobj, struct attribute * attr,
                        const char * buffer, size_t count);

```



典型的, 如 CPU 频率驱动 `cpufreq` (位于 `drivers/cpufreq`) 就是一个 `sysdev_driver` 形式的驱动, 它的主要工作就是提供一些 `sysfs` 的结点, 包括 `cpuinfo_cur_freq`、`cpuinfo_max_freq`、`cpuinfo_min_freq`、`scaling_available_frequencies`、`scaling_available_governors`、`scaling_cur_freq`、`scaling_driver`、`scaling_governor`、`scaling_max_freq`、`scaling_min_freq` 等。用户空间可以手动 `cat`、`echo` 来操作这些结点或者使用 `cpufrequtils` 工具访问这些结点以与内核通信。

还有一类设备虽然不以 `sysdev_driver` 的形式存在, 但是其本质上只是包含 `sysfs` 结点。典型例子包括 I²C EEPROM, 它以 I²C Client 驱动的形式存在 (后续章节会进行介绍, 类似于前文所说的 SPI 外设驱动), 但该驱动 `drivers/i2c/chips/eeprom.c` 通过 `sysfs_create_bin_file()` 创建二进制 `sysfs` 文件, 该二进制结点对应的 `bin_attribute` 如代码清单 12.23 所示。

代码清单 12.23 EEPROM 驱动的 `bin.attribute` 实例

```
1 static struct bin_attribute eeprom_attr = {
2     .attr = {
3         .name = "eeprom",
4         .mode = S_IRUGO,
5     },
6     .size = EEPROM_SIZE,
7     .read = eeprom_read,
8 };
```

之后透过 `/sys` 目录里的 “`eeprom`” 文件即可访问该 EEPROM。创建这个结点的语句是:

```
sysfs_create_bin_file(&client->dev.kobj, &eeprom_attr);
```

其中第 1 个参数是 `bin_attribute` 所对应设备的 `kobject` 指针, 这预示着该 “`eeprom`” 在 `/sys` 中将位于 `client->dev` 这个 `device` 的目录之下。

`drivers/leds/ leds-gpio.c` 的基于 GPIO 的 LED 驱动也提供了 `sysfs` 结点, 针对此驱动, 在 LDD6410 的板文件中只需要定义 LED 对应的 GPIO 信息并作为 `leds-gpio` 这个 `platform_device` 的 `platform_data` 即可:

```
static struct gpio_led ldd6410_leds[] = {
    [0] = {
        .name = "LED1",
        .gpio = S3C64XX_GPM(0),
    },
    [1] = {
        .name = "LED2",
        .gpio = S3C64XX_GPM(1),
    },
    [2] = {
        .name = "LED3",
        .gpio = S3C64XX_GPM(2),
    },
    [3] = {
        .name = "LED4",
        .gpio = S3C64XX_GPM(3),
    },
};

static struct gpio_led_platform_data ldd6410_gpio_led_pdata = {
    .num_leds = ARRAY_SIZE(ldd6410_leds),
    .leds = ldd6410_leds,
};
```

```
static struct platform_device ldd6410_device_led = {
    .name      = "leds-gpio",
    .id        = -1,
    .dev       = {
        .platform_data = &ldd6410_gpio_led_pdata,
    },
};
```

通过如下命令可以点亮 LDD6410 右下角的 LED1:

```
# echo 1 > /sys/devices/platform/leds-gpio/leds\:LED1/brightness
```

熄灭 LED1:

```
# echo 0 > /sys/devices/platform/leds-gpio/leds\:LED1/brightness
```

12.7 Linux 设备驱动的固件加载

一个外设的运行可能依赖于固件, 如一些 CSR 公司的 WiFi 模块, 在启动前需要加载固件。传统的设备驱动将固件的二进制码作为一个数组直接编译进目标代码, 而在 Linux 2.6 中, 有一套成熟的固件加载流程。

首先, 申请固件的驱动程序发起如下请求:

```
int request_firmware(const struct firmware **fw, const char *name, struct device *device);
```

第 1 个参数用于保存申请到的固件, 第 2 个参数是固件名, 第 3 个参数是申请固件的设备结构体。

在发起此调用后, 内核的 udevd 会配合将固件通过对应的 sysfs 结点写入内核 (在设置好 udev 规则的情况下)。之后内核将收到的 firmware 写入外设, 最后通过如下 API 释放请求:

```
void release_firmware(const struct firmware *fw);
```

下面看一个典型的例子 drivers/media/video/cx25840/cx25840-firmware.c 的 cx25840_loadfw() 函数, 如代码清单 12.24 所示。

代码清单 12.24 Linux 设备驱动申请 firmware 的例子

```
1 int cx25840_loadfw(struct i2c_client *client)
2 {
3     struct cx25840_state *state = i2c_get_clientdata(client);
4     const struct firmware *fw = NULL;
5     u8 buffer[FWSEND];
6     const u8 *ptr;
7     int size, retval;
8
9     if (state->is_cx23885)
10         firmware = FWFILE_CX23885;
11     /* 申请 firmware */
12     if (request_firmware(&fw, firmware, FWDEV(client)) != 0) {
13         v4l_err(client, "unable to open firmware %s\n", firmware);
14         return -EINVAL;
15     }
16     /* 开始加载 firmware 到设备 */
17     start_fw_load(client);
18 }
```



```
19  buffer[0] = 0x08;
20  buffer[1] = 0x02;
21
22  size = fw->size;
23  ptr = fw->data;
24  while (size > 0) {
25      int len = min(FWSEND - 2, size);
26
27      memcpy(buffer + 2, ptr, len);
28
29      retval = fw_write(client, buffer, len + 2);
30
31      if (retval < 0) {
32          release_firmware(fw);
33          return retval;
34      }
35
36      size -= len;
37      ptr += len;
38  }
39
40  end_fw_load(client);
41
42  size = fw->size;
43  /* 释放 firmware */
44  release_firmware(fw);
45
46  return check_fw_load(client, size);
47 }
```

12.8 Android 设备驱动

Android 的设备驱动与 Linux 一样，因为 Android 本身基于 Linux 内核，但是 Android 对内核引入了如下主要补丁。

1. binder IPC 系统

binder 机制是 Android 提供了一种进程间通信方法，使一个进程可以以类似远程过程调用的形式调用另一个进程所提供的功能，LDD6410 开发板已经将它的代码移植到 `drivers/android/binder.h`、`drivers/android/binder.c` 下面，从代码清单 12.24 可知，它就是一种典型的以 `miscdevice` 形式实现的字符设备，而且提供了一些 `/proc` 结点。本质上，binder 用户空间的程序绝大多数情况下在底层是调用了 binder 驱动的 `ioctl()` 函数。

代码清单 12.25 Android binder 驱动

```
1  static struct file_operations binder_fops = {
2  .owner = THIS_MODULE,
3  .poll = binder_poll,
4  .unlocked_ioctl = binder_ioctl,
5  .mmap = binder_mmap,
6  .open = binder_open,
```

```

7  .flush = binder_flush,
8  .release = binder_release,
9  };
10
11 static struct miscdevice binder_miscdev = {
12  .minor = MISC_DYNAMIC_MINOR,
13  .name = "binder",
14  .fops = &binder_fops
15 };
16
17 static int __init binder_init(void)
18 {
19  int ret;
20
21  binder_proc_dir_entry_root = proc_mkdir("binder", NULL);
22  if (binder_proc_dir_entry_root)
23      binder_proc_dir_entry_proc = proc_mkdir("proc", binder_proc_dir_entry_root);
24  ret = misc_register(&binder_miscdev);
25  if (binder_proc_dir_entry_root) {
26      create_proc_read_entry("state", S_IRUGO, binder_proc_dir_entry_root, ...);
27      create_proc_read_entry("stats", S_IRUGO, binder_proc_dir_entry_root, ...);
28      create_proc_read_entry("transactions", S_IRUGO, ...);
29      create_proc_read_entry("transaction_log", S_IRUGO, ...);
30      create_proc_read_entry("failed_transaction_log", S_IRUGO, ...);
31  }
32  return ret;
33 }

```

Android 中的 binder 通信基于 Service/Client 模型,所有需要 IBinder 通信的进程都必须创建一个 IBinder 接口。Android 虚拟机启动之前系统会先启动 Service Manager 进程,Service Manager 打开 binder 驱动,并通知 binder 驱动程序这个进程将作为 System Service Manager,然后该进程将进入一个循环,等待处理来自其他进程的数据。

而在用户程序方面,Service 端创建一个 System Service 后,通过 defaultServiceManager()可以得到远程 ServiceManager 的接口,通过这个接口我们可以调用 addService()函数将新的 System Service 添加到 Service Manager 进程中。对于 Client 端而言,则可以通过 getService()获取到需要连接的目的 Service 的 IBinder 对象。对用户程序而言,获得这个对象后就可以通过 binder 驱动访问 Service 对象中的方法。

Client 与 Service 在不同的进程中,通过这种方式实现了类似线程间的迁移的通信方式,对用户程序而言当调用 Service 返回的 IBinder 接口后,访问 Service 中的方法就如同调用自己的函数。两个进程间通信就好像是一个进程进入另一个进程执行代码然后带着执行的结果返回。

2. ashmem 内存共享机制

ashmem 是 Android 新增的一种内存分配/共享机制,LDD6410 开发板已经将它的代码移植到 mm/ashmem.c 下面。从代码清单 12.26 可知,它也是一种典型的以 miscdevice 形式实现的字符设备。

代码清单 12.26 Android ashmem 驱动

```

1  static struct file_operations ashmem_fops = {
2  .owner = THIS_MODULE,
3  .open = ashmem_open,
4  .release = ashmem_release,

```



```
5 .mmap = ashmem_mmap,
6 .unlocked_ioctl = ashmem_ioctl,
7 .compat_ioctl = ashmem_ioctl,
8 };
9
10 static struct miscdevice ashmem_misc = {
11 .minor = MISC_DYNAMIC_MINOR,
12 .name = "ashmem",
13 .fops = &ashmem_fops,
14 };
15
16 static int __init ashmem_init(void)
17 {
18 ...
19
20 ret = misc_register(&ashmem_misc);
21 if (unlikely(ret)) {
22     printk(KERN_ERR "ashmem: failed to register misc device!\n");
23     return ret;
24 }
25
26 ...
27 }
```

在 `dev` 目录下对应的设备是 `/dev/ashmem`，相比于传统的内存分配机制，如 `malloc`、匿名/命名 `mmap`，其好处是提供了辅助内核内存回收算法的 `pin/unpin` 机制。

`ashmem` 的典型用法是先打开设备文件，然后做 `mmap` 映射。

(1) 通过调用 `ashmem_create_region()` 函数打开和设置 `ashmem`，这个函数的实质工作为：

```
fd = open("/dev/ashmem", O_RDWR);
ioctl(fd, ASHMEM_SET_NAME, region_name);
ioctl(fd, ASHMEM_SET_SIZE, region_size);
```

(2) 应用程序一般会调用 `mmap` 来把 `ashmem` 分配的空间映射到进程空间：

```
mapAddr = mmap(NULL, pHdr->mapLength, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);
```

应用程序还可以通过 `ioctl()` 来 `pin` (`ASHMEM_PIN`) 和 `unpin` (`ASHMEM_UNPIN`) 某一段映射的空间，以提示内核的 `page cache` 算法可以把哪些页面回收，这是一般 `mmap` 做不到的。通过 `ASHMEM_GET_PIN_STATUS` 这个 `IOCTL` 可以查询 `pin` 的状态。

3. Android 电源管理

Android 电源管理针对标准 Linux 内核的电源管理进行了一些优化，这部分代码 LDD6410 开发板已经移植到了 `kernel/power/` 目录，主要新增了如下文件：

```
kernel/power/earlysuspend.c
kernel/power/consoleearlysuspend.c
kernel/power/fbearlysuspend.c
kernel/power/wakelock.c
kernel/power/userwakelock.c
```

4. Android Low Memory Killer

Linux 内核本身提供了 OOM (Out Of Memory) 机制，它可以在系统内存不够的情况下主动杀死进程腾出内存。不过 Android 的 Low Memory Killer 相对于 Linux 标准 OOM 机制更加灵活，它可以根据需要杀死进程来释放内存。LDD6410 板已将 Low Memory Killer 移植到 `drivers/android/lowmemorykiller.c` 文件。

5. Android RAM console 和 log 设备

为了辅助调试，Android 增加了用于将内核打印消息保存起来的 RAM console 和用户应用程序可以写入、读取 log 信息的 logger 设备驱动。这两个驱动分别放置在 `drivers/android/ram_console.c` 和 `drivers/android/logger.c` 文件。

RAM console 通过 `register_console()` 被注册，关于这个接口，本书第 14 章会进行详细介绍，而 logger 又是一个典型的 `miscdevice`。

6. Android alarm、timed_gpio 等。

就 Android 系统本身而言，在其上编写设备驱动没有什么神秘的，基本完全按照 Linux 内核本身的框架进行。而 Android 自身引入的这些补丁，曾经有部分进入过 Linux mainline 的 `drivers/staging` 目录，尔后由于缺少维护的原因，被 Greg KH 移除。

12.9 总结

到目前为止，字符设备驱动整个讲解就暂时划上了一个句号。虽然以字符设备为依托进行讲解，但是，第 6~11 章中所描述的关于阻塞与非阻塞、异步通知、轮询、内存与 I/O 访问、并发控制等机制并非只适用于字符设备，对其他的任何设备，都存在同样的问题，也采用完全相同的处理方法。

但是，真实生活中的驱动并非如第 6~11 章的驱动那般简洁，它往往包含了 `platform`、分层、分离、`miscdevice`、`sysfs`、电源管理、固件加载等诸多内容，因此，学习和领悟第 12 章的内容是我们将驱动的理论用于工程开发的必要环节。

LINUX

第13章 Linux 块设备驱动

块设备是与字符设备并列的概念，这两类设备在 Linux 中驱动的结构有较大差异，总体而言，块设备驱动比字符设备驱动要复杂得多，在 I/O 操作上表现出极大的不同，缓冲、I/O 调度、请求队列等都是与块设备驱动相关的概念。本章将详细讲解 Linux 块设备驱动的编程方法。

13.1 节讲解块设备 I/O 操作的特点，分析字符设备与块设备在 I/O 操作上的差异。

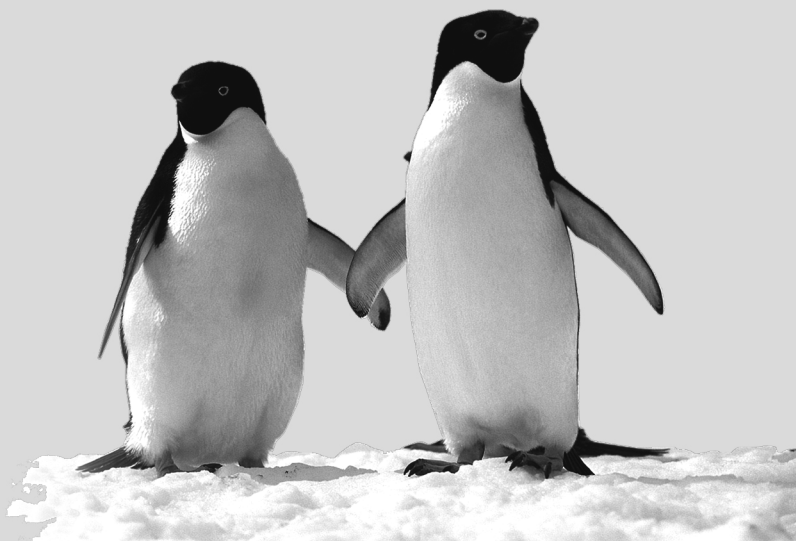
13.2 节从整体上描述 Linux 块设备驱动的结构，分析主要的数据结构、函数及其关系。

13.3~13.5 节分别讲解块设备驱动模块加载与卸载、打开与释放和 `ioctl()` 函数。

13.6 节非常重要，讲述了块设备 I/O 操作所依赖的请求队列的概念及用法。

13.2 节与 13.3~13.6 节是整体与部分的关系，13.2~13.6 节与 13.7 节是迭代递进的关系。

13.7 节在 13.1~13.6 节讲解内容的基础上，总结 Linux 下块设备的读写流程，实现了块设备驱动的一个具体实例，即 `vmem_disk` 的驱动。



13.1 块设备的 I/O 操作特点

字符设备与块设备 I/O 操作的不同如下。

(1) 块设备只能以块为单位接受输入和返回输出，而字符设备则以字节为单位。大多数设备是字符设备，因为它们不需要缓冲而且不以固定块大小进行操作。

(2) 块设备对于 I/O 请求有对应的缓冲区，因此它们可以选择以什么顺序进行响应，字符设备无须缓冲且被直接读写。对于存储设备而言调整读写的顺序作用巨大，因为在读写连续的扇区比分离的扇区更快。

(3) 字符设备只能被顺序读写，而块设备可以随机访问。虽然块设备可随机访问，但是对于磁盘这类机械设备而言，顺序地组织块设备的访问可以提高性能，如图 13.1 所示，对扇区 1、10、3、2 的请求被调整为对扇区 1、2、3、10 的请求。而对 SD 卡、RamDisk 等块设备而言，不存在机械上的原因，进行这样的调整没有必要。

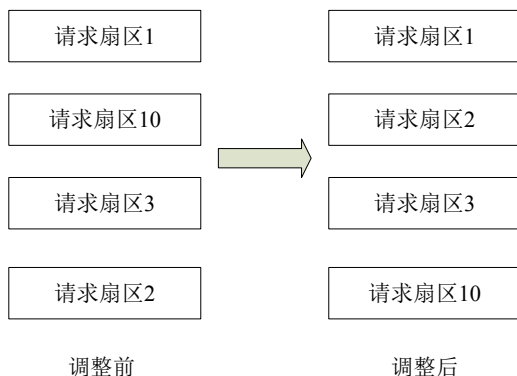


图 13.1 调整块设备 I/O 操作的顺序

13.2 Linux 块设备驱动结构

13.2.1 block_device_operations 结构体

在块设备驱动中，有一个类似于字符设备驱动中 file_operations 结构体的 block_device_operations 结构体，它是对块设备操作的集合，定义如代码清单 13.1 所示。

代码清单 13.1 block_device_operations 结构体

```
1 struct block_device_operations {  
2     int (*open) (struct block_device *, fmode_t);  
3     int (*release) (struct gendisk *, fmode_t);  
4     int (*locked_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
```



```
5     int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
6     int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
7     int (*direct_access) (struct block_device *, sector_t,
8                           void **, unsigned long *);
9     int (*media_changed) (struct gendisk *);
10    int (*revalidate_disk) (struct gendisk *);
11    int (*getgeo) (struct block_device *, struct hd_geometry *);
12    struct module *owner;
13 };
```

下面对其主要的成员函数进行分析。

1. 打开和释放

```
int (*open) (struct gendisk *disk, fmode_t mode);
int (*release) (struct gendisk *disk, fmode_t mode);
```

与字符设备驱动类似，当设备被打开和关闭时将调用它们。

2. I/O 控制

```
int (*ioctl) (struct block_device *bdev, fmode_t mode, unsigned cmd,
              unsigned long arg);
```

上述函数是 `ioctl()` 系统调用的实现，块设备包含大量的标准请求，这些标准请求由 Linux 块设备层处理，因此大部分块设备驱动的 `ioctl()` 函数相当短。

3. 介质改变

```
int (*media_changed) (struct gendisk *gd);
```

被内核调用来检查是否驱动器中的介质已经改变，如果是，则返回一个非 0 值，否则返回 0。这个函数仅适用于支持可移动介质的驱动器，通常需要在驱动中增加一个表示介质状态是否改变的标志变量，非可移动设备的驱动不需要实现这个方法。

4. 使介质有效

```
int (*revalidate_disk) (struct gendisk *gd);
```

`revalidate_disk()` 函数被调用来响应一个介质改变，它给驱动一个机会来进行必要的工作以使新介质准备好。

5. 获得驱动器信息

```
int (*getgeo) (struct block_device *, struct hd_geometry *);
```

该函数根据驱动器的几何信息填充一个 `hd_geometry` 结构体，`hd_geometry` 结构体包含磁头、扇区、柱面等信息，其定义于 `include/linux/hdreg.h` 头文件。

6. 模块指针

```
struct module *owner;
```

一个指向拥有这个结构体的模块的指针，它通常被初始化为 `THIS_MODULE`。

13.2.2 gendisk 结构体

在 Linux 内核中，使用 `gendisk`（通用磁盘）结构体来表示一个独立的磁盘设备（或分区），这个结构体的定义如代码清单 13.2 所示。

代码清单 13.2 gendisk 结构体

```
1 struct gendisk {
2     int major;                                /* 主设备号 */
3     int first_minor;
4     int minors;                               /* 最大的次设备数，如果不能分区，则为 1*/
```

```

5
6     char disk_name[DISK_NAME_LEN];      /* 设备名称 */
7
8     /* 由 partno 索引的分区指针的数组
9      */
10    struct disk_part_tbl *part_tbl;
11    struct hd_struct part0;
12
13    struct block_device_operations *fops; /* 块设备操作集合 */
14    struct request_queue *queue;
15    void *private_data;
16
17    int flags;
18    struct device *driverfs_dev; // FIXME: remove
19    struct kobject *slave_dir;
20
21    struct timer_rand_state *random;
22
23    atomic_t sync_io;                /* RAID */
24    struct work_struct async_notify;
25 #ifdef CONFIG_BLK_DEV_INTEGRITY
26    struct blk_integrity *integrity;
27 #endif
28    int node_id;
29 };

```

major、first_minor 和 minors 共同表征了磁盘的主、次设备号，同一个磁盘的各个分区共享一个主设备号，而次设备号则不同。fops 为 block_device_operations，即上节描述的块设备操作集合。queue 是内核用来管理这个设备的 I/O 请求队列的指针。private_data 可用于指向磁盘的任何私有数据，用法与字符设备驱动 file 结构体的 private_data 类似。hd_struct 成员表示一个分区，而 disk_part_tbl 成员用于容纳分区表，part0 和 part_tbl 二者的关系在于：

```
disk->part_tbl->part[0] = &disk->part0;
```

Linux 内核提供了一组函数来操作 gendisk，如下所示。

1. 分配 gendisk

gendisk 结构体是一个动态分配的结构体，它需要特别的内核操作来初始化，驱动不能自己分配这个结构体，而应该使用下列函数来分配 gendisk：

```
struct gendisk *alloc_disk(int minors);
```

minors 参数是这个磁盘使用的次设备号的数量，一般也就是磁盘分区的数量，此后 minors 不能被修改。

2. 增加 gendisk

gendisk 结构体被分配之后，系统还不能使用这个磁盘，需要调用如下函数来注册这个磁盘设备。

```
void add_disk(struct gendisk *disk);
```

特别要注意的是对 add_disk() 的调用必须发生在驱动程序的初始化工作完成并能响应磁盘的请求之后。

3. 释放 gendisk

当不再需要一个磁盘时，应当使用如下函数释放 gendisk。

```
void del_gendisk(struct gendisk *gp);
```



4. gendisk 引用计数

通过 `get_disk()` 和 `put_disk()` 函数可用来操作 `gendisk` 的引用计数, 这个工作一般不需要驱动亲自做。这两个函数的原型分别为:

```
struct kobject *get_disk(struct gendisk *disk);
void put_disk(struct gendisk *disk);
```

前者最终会调用 “`kobject_get(&disk_to_dev(disk)->kobj);`”, 而后者则会调用 “`kobject_put(&disk_to_dev(disk)->kobj);`”。

13.2.3 request 与 bio 结构体

1. 请求

在 Linux 块设备驱动中, 使用 `request` 结构体来表征等待进行的 I/O 请求, 这个结构体的定义如代码清单 13.3 所示。

代码清单 13.3 request 结构体

```
1 struct request {
2     struct list_head queuelist;
3     struct call_single_data csd;
4     int cpu;
5
6     struct request_queue *q;
7
8     unsigned int cmd_flags;
9     enum rq_cmd_type_bits cmd_type;
10    unsigned long atomic_flags;
11
12    /* 维护 I/O submission 的 BIO 遍历状态
13     * hard_开头的成员仅用于块层内部, 驱动不应该改变它们
14     */
15
16    sector_t sector; /* 要提交的下一个 sector */
17    sector_t hard_sector; /* 要完成的下一个 sector */
18    unsigned long nr_sectors; /* 剩余需要提交的 sector 数 */
19    unsigned long hard_nr_sectors; /* 剩余需要完成的 sector 数 */
20    /* 在当前 segment 中剩余的需提交的 sector 数 */
21    unsigned int current_nr_sectors;
22
23    /* 在当前 segment 中剩余的需完成的 sector 数 */
24    unsigned int hard_cur_sectors;
25
26    struct bio *bio;
27    struct bio *biotail;
28
29    struct hlist_node hash;
30    union {
31        struct rb_node rb_node; /* sort/lookup */
32        void *completion_data;
33    };
34
35    /*
36     * I/O 调度器可获得的两个指针, 如果需要更多, 请动态分配
```

```

37     */
38     void *elevator_private;
39     void *elevator_private2;
40
41     struct gendisk *rq_disk;
42     unsigned long start_time;
43
44     /* scatter-gather DMA 方式下 addr+len 对的数量 (执行物理地址合并后)
45     */
46     unsigned short nr_phys_segments;
47
48     unsigned short ioprio;
49
50     void *special;
51     char *buffer;
52
53     int tag;
54     int errors;
55
56     int ref_count;
57
58     unsigned short cmd_len;
59     unsigned char __cmd[BLK_MAX_CDB];
60     unsigned char *cmd;
61
62     unsigned int data_len;
63     unsigned int extra_len;
64     unsigned int sense_len;
65     void *data;
66     void *sense;
67
68     unsigned long deadline;
69     struct list_head timeout_list;
70     unsigned int timeout;
71     int retries;
72
73     /*
74     * 完成回调函数
75     */
76     rq_end_io_fn *end_io;
77     void *end_io_data;
78
79     struct request *next_rq;
80 };

```

request 结构体的主要成员包括:

```

sector_t hard_sector;
unsigned long hard_nr_sectors;
unsigned int hard_cur_sectors;

```

上述 3 个成员标识还未完成的扇区, `hard_sector` 是第一个尚未传输的扇区, `hard_nr_sectors` 是尚待完成的扇区数, `hard_cur_sectors` 是当前 I/O 操作中待完成的扇区数。

```

sector_t sector;
unsigned long nr_sectors;
unsigned int current_nr_sectors;

```

驱动中会经常与这 3 个成员打交道, 这 3 个成员在内核和驱动交互中发挥着重大作用。它们



以 512 字节大小为一个扇区, 如果硬件的扇区大小不是 512 字节, 则需要进行相应的调整。例如, 如果硬件的扇区大小是 2048 字节, 则在进行硬件操作之前, 需要用 4 来除起始扇区号。

`hard_sector`、`hard_nr_sectors`、`hard_cur_sectors` 与 `sector`、`nr_sectors`、`current_nr_sectors` 之间可认为是“副本”关系。

```
struct bio *bio;
```

`bio` 是这个请求中包含的 `bio` 结构体的链表, 驱动中不宜直接存取这个成员, `__rq_for_each_bio(_bio, rq)`宏封装了对 `bio` 链表的遍历方法, 其定义为:

```
#define __rq_for_each_bio(_bio, rq) \
    if ((rq->bio)) \
        for (_bio = (rq->bio); _bio; _bio = _bio->bi_next)
```

```
char *buffer;
```

指向缓冲区的指针, 数据应当被传送到或者来自这个缓冲区, 这个指针是一个内核虚拟地址, 可被驱动直接引用。

```
unsigned short nr_phys_segments;
```

该值表示相邻的页被合并后, 这个请求在物理内存中占据的段的数目。

如果设备支持分散/聚集 (SG, `scatter/gather`) 操作, 可依据此字段申请 `sizeof(scatterlist)* nr_phys_segments` 的内存, 并使用下列函数进行 DMA 映射:

```
int blk_rq_map_sg(struct request_queue *, struct request *, struct scatterlist *);
```

该函数与 `dma_map_sg()`类似, 它返回 `scatterlist` 列表入口的数量。

```
struct list_head queue_list;
```

用于链接这个请求到请求队列的链表结构, `blkdev_dequeue_request()`可用于从队列中移除请求。

使用如下宏可以从 `request` 获得数据传送的方向。

```
rq_data_dir(struct request *req);
```

0 返回值表示从设备中读, 非 0 返回值表示向设备写。

2. 请求队列

一个块请求队列是一个块 I/O request 的队列, 其定义如代码清单 13.4 所示。

代码清单 13.4 request 队列结构体

```
1 struct request_queue {
2     ...
3     request_fn_proc      *request_fn;
4     make_request_fn      *make_request_fn;
5     prep_rq_fn           *prep_rq_fn;
6     unplug_fn            *unplug_fn;
7     prepare_discard_fn   *prepare_discard_fn;
8     merge_bvec_fn        *merge_bvec_fn;
9     prepare_flush_fn     *prepare_flush_fn;
10    softirq_done_fn       *softirq_done_fn;
11    rq_timed_out_fn       *rq_timed_out_fn;
12    dma_drain_needed_fn   *dma_drain_needed;
13    lld_busy_fn           *lld_busy_fn;
14    ...
15    spinlock_t            __queue_lock;
16    spinlock_t            *queue_lock;
17
18    /*
19     * queue kobject
```



```

20      */
21      struct kobject kobj;
22
23      /*
24       * queue 设置
25       */
26      unsigned long      nr_requests;    /* Max # of requests */
27      unsigned int       nr_congestion_on;
28      unsigned int       nr_congestion_off;
29      unsigned int       nr_batching;
30
31      unsigned int       max_sectors;
32      unsigned int       max_hw_sectors;
33      unsigned short     max_phys_segments;
34      unsigned short     max_hw_segments;
35      unsigned short     hardsect_size;
36      unsigned int       max_segment_size;
37
38      unsigned long      seg_boundary_mask;
39      void               *dma_drain_buffer;
40      unsigned int       dma_drain_size;
41      unsigned int       dma_pad_mask;
42      unsigned int       dma_alignment;
43
44      struct blk_queue_tag *queue_tags;
45      struct list_head   tag_busy_list;
46
47      unsigned int       nr_sorted;
48      unsigned int       in_flight;
49
50      unsigned int       rq_timeout;
51      struct timer_list  timeout;
52      struct list_head   timeout_list;
53
54      /*
55       * sg stuff
56       */
57      unsigned int       sg_timeout;
58      unsigned int       sg_reserved_size;
59      int                node;
60 #ifdef CONFIG_BLK_DEV_IO_TRACE
61      struct blk_trace   *blk_trace;
62 #endif
63      ...
64 };

```

请求队列跟踪等候的块 I/O 请求，它存储用于描述这个设备能够支持的请求的类型信息、它们的最大大小、多少不同的段可进入一个请求、硬件扇区大小、对齐要求等参数，其结果是：如果请求队列被配置正确了，它不会交给该设备一个不能处理的请求。

请求队列还实现一个插入接口，这个接口允许使用多个 I/O 调度器，I/O 调度器（也称电梯）的工作是以最优性能的方式向驱动提交 I/O 请求。大部分 I/O 调度器累积批量的 I/O 请求，并将它们排列为递增（或递减）的块索引顺序后提交给驱动。进行这些工作的原因在于，对于磁头而言，当给定顺序排列的请求时，可以使得磁盘顺序地从头到另一头工作，非常像一个满载的电梯，在一个方向移动直到所有它的“请求”被满足。

另外，I/O 调度器还负责合并邻近的请求，当一个新 I/O 请求被提交给调度器后，它会在队列里



搜寻包含邻近扇区的请求。如果找到一个, 并且如果结果的请求不是太大, 调度器将合并这两个请求。

对磁盘等块设备进行 I/O 操作顺序的调度类似于电梯的原理, 先服务完上楼的乘客, 再服务下楼的乘客效率会更高, 而顺序响应用户的请求则电梯会无序地忙乱。

Linux 2.6 内核包含 4 个 I/O 调度器, 它们分别是 No-op I/O scheduler、Anticipatory I/O scheduler、Deadline I/O scheduler 与 CFQ I/O scheduler。

Noop I/O scheduler 是一个简化的调度程序, 该算法实现了一个简单 FIFO 队列, 它只作最基本的合并与排序。

Anticipatory I/O scheduler 算法推迟 I/O 请求, 以期能对它们进行排序, 获得最高的效率。在每次处理完读请求之后, 不是立即返回, 而是等待几个微妙。在这段时间内, 任何来自临近区域的请求都被立即执行。超时以后, 继续原来的处理。

Deadline I/O scheduler 是针对 Anticipatory I/O scheduler 的缺点进行改善而来的, 它试图把每次请求的延迟降至最低, 该算法重排了请求的顺序来提高性能。它使用轮询的调度器, 简洁小巧, 提供了最小的读取延迟和尚佳的吞吐量, 特别适合于读取较多的环境 (比如数据库)。

CFQ I/O scheduler 为系统内的所有任务分配均匀的 I/O 带宽, 提供一个公平的工作环境, 在多媒体应用中, 能保证 audio、video 及时从磁盘读取数据。

内核 block 目录中的 noop-iosched.c、as-iosched.c、deadline-iosched.c 和 cfq-iosched.c 文件分别实现了上述调度算法。

可以通过给 kernel 添加启动参数, 选择使用的 IO 调度算法, 如:

```
kernel elevator=deadline
```

(1) 初始化请求队列。

```
request_queue_t *blk_init_queue(request_fn_proc *rfn, spinlock_t *lock);
```

该函数的第一个参数是请求处理函数的指针, 第二个参数是控制访问队列权限的自旋锁, 这个函数会发生内存分配的行为, 它可能会失败, 因此一定要检查它的返回值。这个函数一般在块设备驱动模块加载函数中调用。

(2) 清除请求队列。

```
void blk_cleanup_queue(request_queue_t *q);
```

这个函数完成将请求队列返回给系统的任务, 一般在块设备驱动模块卸载函数中调用。

而 blk_put_queue() 宏则定义为:

```
#define blk_put_queue(q) blk_cleanup_queue((q))
```

(3) 分配“请求队列”。

```
request_queue_t *blk_alloc_queue(int gfp_mask);
```

对于 Flash、RAM 盘等完全随机访问的非机械设备, 并不需要进行复杂的 I/O 调度, 这个时候, 应该使用上述函数分配一个“请求队列”, 并使用如下函数来绑定请求队列和“制造请求”函数 (make_request_fn)。

```
void blk_queue_make_request(request_queue_t *q, make_request_fn *mfn);
```

在 13.6.2 节我们会看到, 这种方式分配的“请求队列”实际上不包含任何 request, 所以给其加上引号。

(4) 提取请求。

```
struct request *elv_next_request(struct request_queue *q);
```

上述函数用于返回下一个要处理的请求 (由 I/O 调度器决定), 如果没有请求则返回 NULL。

`elv_next_request()`不会清除请求，它仍然将这个请求保留在队列上，但是标识它为活动的，这个标识将阻止 I/O 调度器合并其他的请求到已开始执行的请求。因为 `elv_next_request()`不从队列里清除请求，因此连续调用它两次，两次会返回同一个请求结构体。

(5) 去除请求。

```
void blkdev_dequeue_request(struct request *req);
```

上述函数从队列中去除一个请求。如果驱动中同时从同一个队列中操作了多个请求，它必须以这样的方式将它们从队列中去除。

如果要将一个已经出列的请求归还到队列中，可以进行以下调用：

```
void elv_requeue_request(request_queue_t *queue, struct request *req);
```

另外，块设备层还提供了一套函数，这些函数可被驱动用来控制一个请求队列的操作，主要包括以下操作。

(6) 启停请求队列。

```
void blk_stop_queue(request_queue_t *queue);
```

```
void blk_start_queue(request_queue_t *queue);
```

如果块设备到达不能处理等候的命令的状态，应调用 `blk_stop_queue()`来告知块设备层。之后，请求函数将不被调用，除非再次调用 `blk_start_queue()`将设备恢复到可处理请求的状态。

(7) 参数设置。

```
void blk_queue_max_sectors(request_queue_t *queue, unsigned short max);
```

```
void blk_queue_max_phys_segments(request_queue_t *queue, unsigned short max);
```

```
void blk_queue_max_hw_segments(request_queue_t *queue, unsigned short max);
```

```
void blk_queue_max_segment_size(request_queue_t *queue, unsigned int max);
```

这些函数用于设置描述块设备可处理的请求的参数。`blk_queue_max_sectors()`描述任一请求可包含的最大扇区数，默认值为 255；`blk_queue_max_phys_segments()`和 `blk_queue_max_hw_segments()`都控制一个请求中可包含的最大物理段（系统内存中不相邻的区），`blk_queue_max_hw_segments()`考虑了系统 I/O 内存管理单元的重映射，这两个参数缺省都是 128。`blk_queue_max_segment_size`告知内核请求段的最大字节数，默认值为 65536。

(8) 通告内核。

```
void blk_queue_bounce_limit(request_queue_t *queue, u64 dma_addr);
```

该函数用于告知内核块设备执行 DMA 时可使用的最高物理地址 `dma_addr`，如果一个请求包含超出这个限制的内存引用，系统将会给这个操作分配一个“反弹”缓冲区。这种方式的代价昂贵，因此应尽量避免使用。

可以给 `dma_addr` 参数提供任何可能的值或使用预先定义的宏，如 `BLK_BOUNCE_HIGH`（对高端内存页使用反弹缓冲区）、`BLK_BOUNCE_ISA`（驱动只可在 16MB 的 ISA 区执行 DMA）或者 `BLK_BOUNCE_ANY`（驱动可在任何地址执行 DMA），缺省值是 `BLK_BOUNCE_HIGH`。

```
blk_queue_segment_boundary(request_queue_t *queue, unsigned long mask);
```

如果我们正在驱动编写的设备无法处理跨越一个特殊大小内存边界的请求，应该使用这个函数来告知内核这个边界。例如，如果设备处理跨 4MB 边界的请求有困难，应该传递一个 `0x3ffff` 掩码，缺省的掩码是 `0xffffffff`（对应 4GB 边界）。

```
void blk_queue_dma_alignment(request_queue_t *queue, int mask);
```

告知内核块设备施加于 DMA 传送的内存对齐限制，所有请求都匹配这个对齐，缺省的屏蔽是 `0x1ff`，它导致所有的请求被对齐到 512 字节边界。



```
void blk_queue_hardsect_size(request_queue_t *queue, unsigned short max);
```

该函数告知内核块设备硬件扇区的大小，所有由内核产生的请求都是这个大小的倍数并且被正确对界。但是，内核块设备层和驱动之间的通信还是以 512 字节扇区为单位进行。

3. 块 I/O

通常一个 bio 对应一个上层传递给块层的 I/O 请求，代码清单 13.5 给出了 bio 结构体的定义。I/O 调度算法可将连续的 bio 合并成一个 request。request 是 bio 经由块层进行调整后的结果，这是 request 和 bio 的区别。所以，一个 request 可以包含多个 bio。

代码清单 13.5 bio 结构体

```
1 struct bio {
2     sector_t bi_sector; /* 要传输的第一个扇区 */
3     struct bio *bi_next; /* 下一个 bio */
4     struct block_device *bi_bdev;
5     unsigned long bi_flags; /* 状态、命令等 */
6     unsigned long bi_rw; /* 低位表示 READ/WRITE, 高位表示优先级 */
7
8     unsigned short bi_vcnt; /* bio_vec 数量 */
9     unsigned short bi_idx; /* 当前 bvl_vec 索引 */
10
11     /* 执行物理地址合并后 segment 的数目 */
12     unsigned short bi_phys_segments;
13
14     unsigned int bi_size;
15
16     /* 为了明了最大的 segment 尺寸, 我们考虑这个 bio 中第一个和最后一个
17     可合并的 segment 的尺寸 */
18     unsigned int bi_hw_front_size;
19     unsigned int bi_hw_back_size;
20
21     unsigned int bi_max_vecs; /* 我们能持有的最大 bvl_vecs 数 */
22     unsigned int bi_comp_cpu; /* completion CPU */
23
24     struct bio_vec *bi_io_vec; /* 实际的 vec 列表 */
25
26     bio_end_io_t *bi_end_io;
27     atomic_t bi_cnt;
28
29     void *bi_private;
30 #if defined(CONFIG_BLK_DEV_INTEGRITY)
31     struct bio_integrity_payload *bi_integrity; /* 数据完整性 */
32 #endif
33
34     bio_destructor_t *bi_destructor; /* 析构 */
35 };
```

下面我们对其中的核心成员进行分析：

```
sector_t bi_sector;
```

标识这个 bio 要传送的第一个（512 字节）扇区。

```
unsigned int bi_size;
```

被传送的数据大小，以字节为单位，驱动中可以使用 `bio_sectors(bio)` 宏获得以扇区为单位的大小，该宏实际定义为 “`((bio)->bi_size >> 9)`”。

```
unsigned long bi_flags;
unsigned long bi_rw;
```

一组描述 bio 的标志，如果这是一个写请求，bi_rw 最低有效位被置位，可以使用 `bio_data_`

dir(bio)宏来获得读写方向，该宏实际定义为“((bio)->bi_rw & 1)”。

bio的核心是一个称为 bi_io_vec 的数组，它由 bio_vec 结构体组成，bio_vec 结构体的定义如代码清单 13.6 所示。

代码清单 13.6 bio_vec 结构体

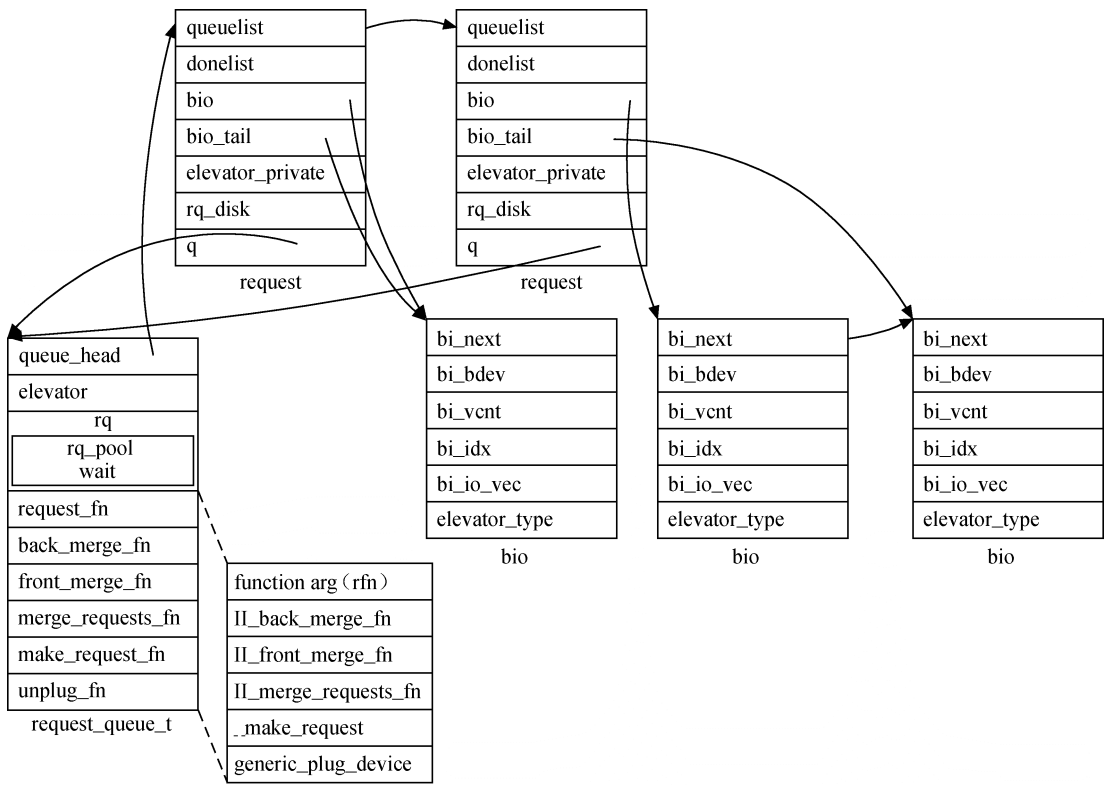
```
1 struct bio_vec {
2     struct page *bv_page; /* 页指针 */
3     unsigned int bv_len; /* 传输的字节数 */
4     unsigned int bv_offset; /* 偏移位置 */
5 };
```

我们不应该直接访问 bio 的 bio_vec 成员，而应该使用 bio_for_each_segment()宏来进行这项工作，可以用这个宏循环遍历整个 bio 中的每个段，这个宏的定义如代码清单 13.7 所示。

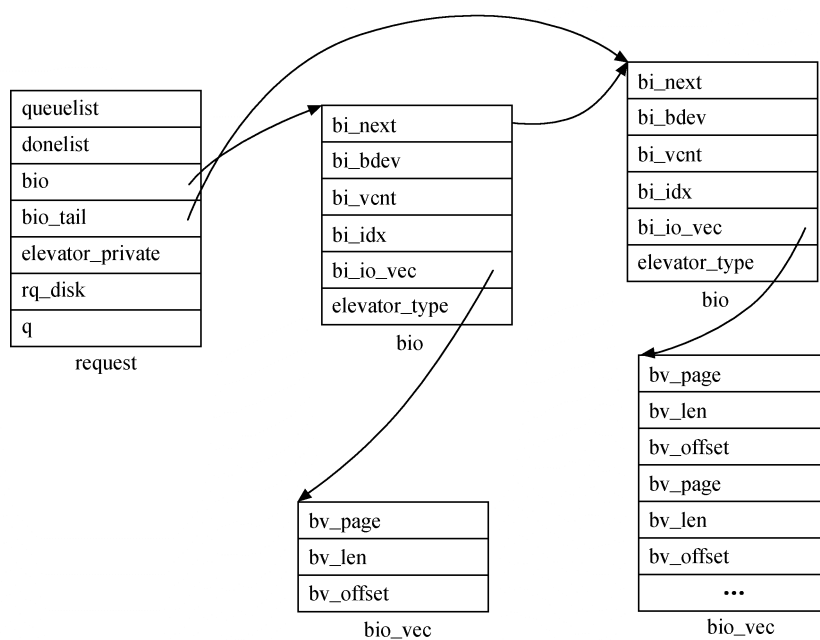
代码清单 13.7 bio_for_each_segment 宏

```
1 #define __bio_for_each_segment(bvl, bio, i, start_idx) \
2     for (bvl = bio_iovec_idx((bio), (start_idx)), i = (start_idx); \
3         i < (bio)->bi_vcnt; \
4         bvl++, i++)
5
6 #define bio_for_each_segment(bvl, bio, i) \
7     __bio_for_each_segment(bvl, bio, i, (bio)->bi_idx)
```

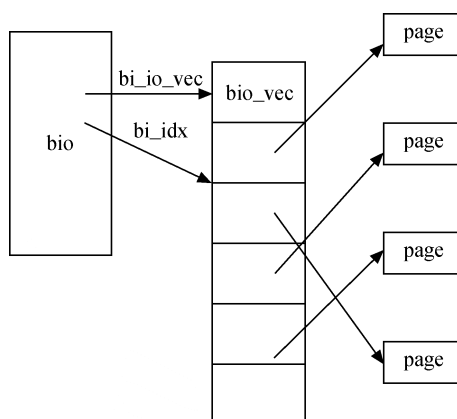
图 13.2 (a) 所示为 request 队列、request 与 bio 数据结构之间的关系，13.2 (b) 所示为 request、bio 和 bio_vec 数据结构之间的关系，13.2 (c) 所示为 bio 与 bio_vec 数据结构之间的关系，因此整个图 13.2 递归地呈现了 request 队列、request、bio 和 bio_vec 这 4 个结构体之间的关系。



(a) request 与 bio



(b) request、bio 和 bio_vec



(c) bio 与 bio_vec

图 13.2 request 队列、request、bio 和 bio_vec 结构体之间的关系

内核还提供了一组函数（宏）用于操作 bio：

```
int bio_data_dir(struct bio *bio);
```

这个函数可用于获得数据传输的方向是 READ 还是 WRITE。

```
struct page *bio_page(struct bio *bio);
```

这个函数可用于获得目前的页指针。

```
int bio_offset(struct bio *bio);
```

这个函数返回操作对应的当前页内的偏移，通常块 I/O 操作本身就是页对齐的。

```
int bio_cur_sectors(struct bio *bio);
```

这个函数返回当前 bio_vec 要传输的扇区数。

```
char *bio_data(struct bio *bio);
```

这个函数返回数据缓冲区的内核虚拟地址。

```
char *bvec_kmap_irq(struct bio_vec *bvec, unsigned long *flags);
```

这个函数返回一个内核虚拟地址，这个地址可用于存取被给定的 `bio_vec` 入口指向的数据缓冲区。它也会屏蔽中断并返回一个原子 `kmap`（用于高端内存映射），因此，在 `bvec_kunmap_irq()` 被调用以前，驱动不应该睡眠。

```
void bvec_kunmap_irq(char *buffer, unsigned long *flags);
```

这个函数是 `bvec_kmap_irq()` 函数的“反函数”，它撤销 `bvec_kmap_irq()` 创建的映射。

```
char *bio_kmap_irq(struct bio *bio, unsigned long *flags);
```

这个函数是对 `bvec_kmap_irq()` 的包装，它返回给定的 `bio` 的当前 `bio_vec` 入口的映射。

```
char *_bio_kmap_atomic(struct bio *bio, int i, enum km_type type);
```

这个函数通过 `kmap_atomic()` 获得返回给定 `bio` 的第 `i` 个缓冲区的虚拟地址。

```
void __bio_kunmap_atomic(char *addr, enum km_type type);
```

这个函数返还由 `_bio_kmap_atomic()` 获得的内核虚拟地址。

需要注意的是，`xx_kmap_xx`、`xx_kunmap_xx` 系列函数针对的是支持高端内存的驱动。

另外，对 `bio` 的引用计数通过如下宏/函数完成：

```
#define bio_get(bio)    atomic_inc(&(bio)->bi_cnt)
void bio_put(struct bio *bio); /* 释放对 bio 的引用 */
```

如下函数用于在内核中向块层提交一个 BIO：

```
void submit_bio(int rw, struct bio *bio);
```

结合使用 `bio_get()`、`submit_bio()`、`bio_put()` 的流程一般是：

```
bio_get(bio);
submit_bio(rw, bio);
if (bio->bi_flags ...)
    do_something
bio_put(bio);
```

13.2.4 块设备驱动注册与注销

块设备驱动中的第一个工作通常是注册它们自己到内核，完成这个任务的函数是 `register_blkdev()`，其原型为：

```
int register_blkdev(unsigned int major, const char *name);
```

`major` 参数是块设备要使用的主设备号，`name` 为设备名，它会在 `/proc/devices` 中被显示。如果 `major` 为 0，内核会自动分配一个新的主设备号，`register_blkdev()` 函数的返回值就是这个主设备号。如果 `register_blkdev()` 返回一个负值，表明发生了一个错误。

与 `register_blkdev()` 对应的注销函数是 `unregister_blkdev()`，其原型为：

```
int unregister_blkdev(unsigned int major, const char *name);
```

这里，传递给 `register_blkdev()` 的参数必须与传递给 `unregister_blkdev()` 的参数匹配，否则这个函数返回 `-EINVAL`。

值得一提的是，在 Linux 2.6 内核中，对 `register_blkdev()` 的调用完全是可选的，`register_blkdev()` 的功能已随时间正在减少，这个调用最多只完成两件事。

- ① 如果需要，分配一个动态主设备号。
- ② 在 `/proc/devices` 中创建一个入口。

在将来的内核中，`register_blkdev()` 可能会被去掉。但是目前的大部分驱动仍然调用它。代码清单 13.8 给出了一个块设备驱动注册的模板。



代码清单 13.8 块设备驱动注册模板

```
1 xxx_major = register_blkdev(xxx_major, "xxx");
2 if (xxx_major <= 0) { /* 注册失败 */
3     printk(KERN_WARNING "xxx: unable to get major number\n");
4     return -EBUSY;
5 }
```

13.3 Linux 块设备驱动的模块加载与卸载

在块设备驱动的模块加载函数中通常需要完成如下工作。

- ① 分配、初始化请求队列，绑定请求队列和请求函数。
- ② 分配、初始化 gendisk，给 gendisk 的 major、fops、queue 等成员赋值，最后添加 gendisk。
- ③ 注册块设备驱动。

代码清单 13.9 和 13.10 分别给出了使用 blk_alloc_queue() 分配请求队列并使用 blk_queue_make_request() 绑定“请求队列”和“制造请求”的函数，以及使用 blk_init_queue() 初始化请求队列并绑定请求队列与请求处理函数两种不同情况下的块设备驱动模块加载函数模板。

代码清单 13.9 块设备驱动的模块加载函数模板（使用 blk.alloc.queue）

```
1 static int __init xxx_init(void)
2 {
3     /* 分配 gendisk */
4     xxx_disks = alloc_disk(1);
5     if (!xxx_disks)
6         goto out;
7
8     /* 块设备驱动注册 */
9     if (register_blkdev(XXX_MAJOR, "xxx")) {
10         err = - EIO;
11         goto out;
12     }
13
14     /* “请求队列”分配 */
15     xxx_queue = blk_alloc_queue(GFP_KERNEL);
16     if (!xxx_queue)
17         goto out_queue;
18
19     blk_queue_make_request(xxx_queue, &xxx_make_request); /* 绑定“制造请求”函数 */
20     blk_queue_hardsect_size(xxx_queue, xxx_blocksize); /* 硬件扇区尺寸设置 */
21
22     /* gendisk 初始化 */
23     xxx_disks->major = XXX_MAJOR;
24     xxx_disks->first_minor = 0;
25     xxx_disks->fops = &xxx_op;
26     xxx_disks->queue = xxx_queue;
27     sprintf(xxx_disks->disk_name, "xxx%d", i);
28     add_disk(xxx_disks); /* 添加 gendisk */
29 }
```



```

30  return 0;
31  out_queue: unregister_blkdev(XXX_MAJOR, "xxx");
32  out: put_disk(xxx_disks);
33  blk_cleanup_queue(xxx_queue);
34
35  return -ENOMEM;
36 }

```

代码清单 13.10 块设备驱动的模块加载函数模板（使用 blk.init.queue）

```

1  static int _init xxx_init(void)
2  {
3      /* 块设备驱动注册 */
4      if (register_blkdev(XXX_MAJOR, "xxx")) {
5          err = - EIO;
6          goto out;
7      }
8
9      /* 请求队列初始化 */
10     xxx_queue = blk_init_queue(xxx_request, xxx_lock);
11     if (!xxx_queue)
12         goto out_queue;
13
14     blk_queue_hardsect_size(xxx_queue, xxx_blocksize); /* 硬件扇区尺寸设置 */
15
16     /* gendisk 初始化 */
17     xxx_disks->major = XXX_MAJOR;
18     xxx_disks->first_minor = 0;
19     xxx_disks->fops = &xxx_op;
20     xxx_disks->queue = xxx_queue;
21     sprintf(xxx_disks->disk_name, "xxx%d", i);
22     set_capacity(xxx_disks, xxx_size * 2);
23     add_disk(xxx_disks); //添加 gendisk
24
25     return 0;
26     out_queue: unregister_blkdev(XXX_MAJOR, "xxx");
27     out: put_disk(xxx_disks);
28     blk_cleanup_queue(xxx_queue);
29
30     return - ENOMEM;
31 }

```

在块设备驱动的模块卸载函数中完成与模块加载函数相反的工作。

- ① 清除请求队列。
- ② 删除 gendisk 和对 gendisk 的引用。
- ③ 删除对块设备的引用，注销块设备驱动。

代码清单 13.11 给出了块设备驱动的模块卸载函数的模板。

代码清单 13.11 块设备驱动的模块卸载函数模板

```

1  static void _exit xxx_exit(void)
2  {
3      if (bdev) {
4          invalidate_bdev(xxx_bdev, 1);
5          blkdev_put(xxx_bdev);

```



```
6 }
7 del_gendisk(xxx_disks); /* 删除 gendisk */
8 put_disk(xxx_disks);
9 blk_cleanup_queue(xxx_queue[i]); /* 清除请求队列 */
10 unregister_blkdev(XXX_MAJOR, "xxx");
11 }
```

13.4 块设备的打开与释放

块设备驱动的 `open()` 和 `release()` 函数并非是必须的, 一个简单的块设备驱动可以不提供 `open()` 和 `release()` 函数。

块设备驱动的 `open()` 函数和其字符设备驱动的对等体不太相似, 不以相关的 `inode` 和 `file` 结构体指针作为参数。在 `open()` 中我们可以通过 `block_device` 参数 `bdev` 获取 `private_data`、在 `release()` 函数中则通过 `gendisk` 参数 `disk` 获取, 如代码清单 13.12 所示。

代码清单 13.12 在块设备的 `open()/release()` 函数中获取 `private_data`

```
1 static int xxx_open(struct block_device *bdev, fmode_t mode)
2 {
3     struct xxx_dev *dev = bdev->bd_disk->private_data;
4
5     ...
6
7     return 0;
8 }
9
10 static int xxx_release(struct gendisk *disk, fmode_t mode)
11 {
12     struct xxx_dev *dev = disk->private_data;
13
14     ...
15
16     return 0;
17 }
```

在一个处理真实的硬件设备的驱动中, `open()` 和 `release()` 方法还应当设置驱动和硬件的状态, 这些工作可能包括启停磁盘、加锁一个可移出设备和分配 DMA 缓冲等。

13.5 块设备驱动的 `ioctl` 函数

与字符设备驱动一样, 块设备可以包含一个 `ioctl()` 函数以提供对设备的 I/O 控制能力。实际上, 高层的块设备层代码处理了绝大多数 I/O 控制, 如 `BLKFLSBUF`、`BLKROSET`、`BLKDISCARD`、`HDIO_GETGEO`、`BLKROGET`、`BLKSECTGET` 等, 因此, 具体的块设备驱动中通常只需要实现设备相关的 `ioctl` 命令。例如, 源代码文件为 `drivers/block/floppy.c` 中实现了与软驱相关的命令如 `FDEJECT`、`FDSETPRM`、`FDFMTTRK` 等。

13.6 块设备驱动的 I/O 请求处理

13.6.1 使用请求队列

块设备驱动请求函数的原型为：

```
void request(request_queue_t *queue);
```

这个函数不能由驱动自己调用，只有当内核认为是时候让驱动处理对设备的读写等操作时，它才调用这个函数。

请求函数可以在没有完成请求队列中的所有请求的情况下返回，甚至在一个请求不完成都可以返回。但是，对大部分设备而言，在请求函数中处理完所有请求后再返回通常是值得推荐的方法。代码清单 13.14 给出了一个简单的 request() 函数的例子。

代码清单 13.13 块设备驱动请求函数例程

```
1 static void xxx_request(request_queue_t *q)
2 {
3     struct request *req;
4     while ((req = elv_next_request(q)) != NULL) {
5         struct xxx_dev *dev = req->rq_disk->private_data;
6         if (!blk_fs_request(req)) { /* 不是文件系统请求 */
7             printk(KERN_NOTICE "Skip non-fs request\n");
8             end_request(req, 0);
9             continue;
10        }
11        xxx_transfer(dev, req->sector, req->current_nr_sectors, req->buffer,
12                    rq_data_dir(req)); /* 处理这个请求 */
13        end_request(req, 1); /* 通知成功完成这个请求 */
14    }
15 }
16
17 /* 完成具体的块设备 I/O 操作 */
18 static void xxx_transfer(struct xxx_dev *dev, unsigned long sector, unsigned
19 long nsect, char *buffer, int write)
20 {
21     unsigned long offset = sector * KERNEL_SECTOR_SIZE;
22     unsigned long nbytes = nsect * KERNEL_SECTOR_SIZE;
23     if ((offset + nbytes) > dev->size) {
24         printk(KERN_NOTICE "Beyond-end write (%ld %ld)\n", offset, nbytes);
25         return ;
26     }
27     if (write)
28         write_dev(offset, buffer, nbytes); /* 向设备写 nbytes 个字节的数据 */
29     else
30         read_dev(offset, buffer, nbytes); /* 从设备读 nbytes 个字节的数据 */
31 }
```

上述代码第 4 行使用 elv_next_request() 获得队列中第一个未完成的请求，end_request() 会将请求从请求队列中剥离。第 6 行判断请求是否为文件系统请求，如果不是，则直接清除，调用 end_



request(), 传递给 end_request() 的第二个参数为 0 意味着处理该请求失败。而第 13 行传递给 end_request() 的第二个参数为 1 意味着该请求处理成功。

end_request() 函数非常重要, 其源代码如代码清单 13.14 所示。不过要留意的是, 新的内核版本建议在驱动中调用 blk_end_request() 或 __blk_end_request() 来结束 request。

代码清单 13.14 end_request() 函数源代码

```
1 void end_request(struct request *req, int uptodate)
2 {
3     int error = 0;
4
5     if (uptodate <= 0)
6         error = uptodate ? uptodate : -EIO;
7
8     __blk_end_request(req, error, req->hard_cur_sectors << 9);
9 }
10 int __blk_end_request(struct request *rq, int error, unsigned int nr_bytes)
11 {
12     if (rq->bio && __end_that_request_first(rq, error, nr_bytes))
13         return 1;
14
15     add_disk_randomness(rq->rq_disk);
16
17     end_that_request_last(rq, error);
18
19     return 0;
20 }
```

当设备已经完成一个 I/O 请求的部分或者全部扇区传输后, 它必须通告块设备层, 上述代码中的第 12 和 17 行完成这个工作。__end_that_request_first() 函数的原型为:

```
int __end_that_request_first(struct request *req, int error,
                           int nr_bytes)
```

这个函数告知块设备层, 块设备驱动已经完成 nr_bytes 个扇区的传送。__end_that_request_first() 的返回值是一个标志, 指示是否这个请求中的所有扇区已经被传送。返回值为 0 表示所有的扇区已经被传送并且这个请求完成, 之后, 我们必须使用 blkdev_dequeue_request() 来从队列中清除这个请求。最后, 将这个请求传递给 end_that_request_last() 函数。

```
void end_that_request_last(struct request *req);
```

end_that_request_last() 通知所有正在等待这个请求完成的对象请求已经完成并回收这个请求结构体。

第 15 行的 add_disk_randomness() 函数的作用是使用块 I/O 请求的时间信息来给系统的随机数池贡献熵, 它不影响块设备驱动。但是, 仅当磁盘的操作时间是真正随机的时候 (gendisk 的 random 成员不为 0, 大部分机械设备如此), 它才会调用 add_timer_randomness()。

代码清单 13.15 给出了一个更复杂的请求函数, 它进行了 3 层遍历: 遍历请求队列中的每个请求, 遍历请求中的每个 bio, 遍历 bio 中的每个段。

代码清单 13.15 请求函数遍历请求、bio 和段

```
1 static void xxx_full_request(request_queue_t *q)
2 {
3     struct request *req;
```

```

4  int sectors_xferred;
5  struct xxx_dev *dev = q->queuedata;
6  /* 遍历每个请求 */
7  while ((req = elv_next_request(q)) != NULL) {
8      if (!blk_fs_request(req)) {
9          printk(KERN_NOTICE "Skip non-fs request\n");
10         continue;
11     }
12     end_request(req, 0);
13     continue;
14     sectors_xferred = xxx_xfer_request(dev, req);
15     end_request(req, 1); /* 通知成功完成这个请求 */
16 }
17 /* 请求处理 */
18 static int xxx_xfer_request(struct xxx_dev *dev, struct request *req)
19 {
20     struct bio_vec *bvec;
21     /* 遍历请求中的每个 bio 的每个 segment */
22     rq_for_each_segment(bvec, req, iter) {
23         ...
24     }
25 }

```

上述代码中第 26 行调用的 `rq_for_each_segment()` 实际是一个二重循环，它的第一重循环遍历一个 `request` 的每个 `bio`，第二重循环遍历一个 `bio` 的每个 `segment`：

```

#define rq_for_each_segment(bvl, _rq, _iter) \
    __rq_for_each_bio(_iter.bio, _rq) \
        bio_for_each_segment(bvl, _iter.bio, _iter.i)

```

图 13.3 所示为一个请求队列内 `request`、`bio` 以及 `bio` 中 `segment` 的层层遍历关系。

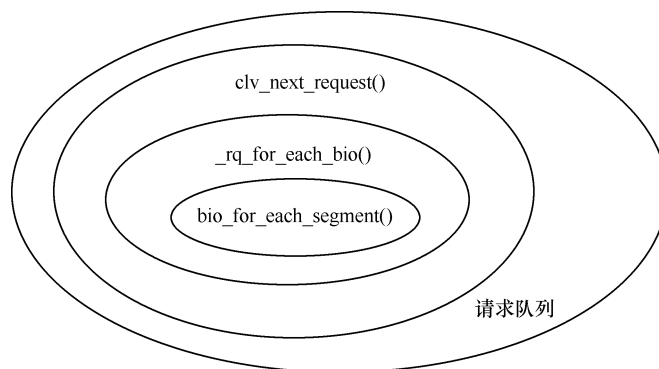


图 13.3 遍历一个请求队列

13.6.2 不使用请求队列

使用请求队列对于一个机械的磁盘设备而言的确有助于提高系统的性能，但是对于许多块设备，如数码相机的存储卡、RAM 盘等完全可真正随机访问的设备而言，无法从高级的请求队列逻辑中获益。对于这些设备，块层支持“无队列”的操作模式，为使用这个模式，驱动必须提供一个“制造请求”函数，而不是一个请求函数，“制造请求”函数的原型为：

```

typedef int (make_request_fn) (request_queue_t *q, struct bio *bio);

```



上述函数的第一个参数仍然是“请求队列”，但是这个“请求队列”实际不包含任何 request，因为块层没有必要将 bio 调整为 request。因此，“制造请求”函数的主要参数是 bio 结构体，这个 bio 结构体表示一个或多个要传送的缓冲区。“制造请求”函数或者直接进行传输，或者把请求重定向给其他设备。

在“制造请求”函数中处理 bio 的方式与 13.6.1 小节中讲解的完全一致，但是在处理完成后应该使用 bio_endio() 函数通知处理结束，如下所示：

```
void bio_endio(struct bio *bio, unsigned int bytes, int error);
```

参数 bytes 是已经传送的字节数，它可以比这个 bio 所代表的字节数少，这意味着“部分完成”，同时 bio 结构体中的当前缓冲区指针需要更新。当设备进一步处理这个 bio 后，驱动应该再次调用 bio_endio()，如果不能完成这个请求，应指出一个错误，错误码赋值给 error 参数。

不管对应的 I/O 处理成功与否，“制造请求”函数都应该返回 0。如果“制造请求”函数返回一个非零值，bio 将被再次提交。

代码清单 13.16 所示为一个“制造请求”函数的例子。

代码清单 13.16 “制造请求”函数例程

```
1 static int xxx_make_request(request_queue_t *q, struct bio *bio)
2 {
3     struct xxx_dev *dev = q->queuedata;
4     int status;
5     status = xxx_xfer_bio(dev, bio); /* 处理 bio */
6     bio_endio(bio, bio->bi_size, status); /* 通告结束 */
7     return 0;
8 }
```

为了使用无队列的 I/O 请求处理，驱动模块的加载函数应遵循代码清单 13.9 的模板而非代码清单 13.10 的模板，而使用请求队列时，驱动模块的加载函数应遵循代码清单 13.10 的模板。

13.7 实例 1：vmem_disk 驱动

13.7.1 vmem_disk 的硬件原理

vmem_disk 是一种模拟磁盘，其数据实际上存储在 RAM 中。它使用通过 vmalloc() 分配出来的内存空间来模拟出一个磁盘，以块设备的方式来访问这片内存。该驱动是对字符设备驱动相应章节 globalmem 驱动的块方式改造。

在加载 vmem_disk.ko 后，使用默认模块参数的情况下，系统会增加 4 个块设备结点：

```
root@lihacker-laptop:~/develop/svn/ldd6410-read-only/training/kernel/drivers/vmem_d
isk# ls -l /dev/vmem_disk*
brw-rw---- 1 root disk 251, 0 2010-04-18 11:53 /dev/vmem_diska
brw-rw---- 1 root disk 251, 16 2010-04-18 11:52 /dev/vmem_diskb
brw-rw---- 1 root disk 251, 32 2010-04-18 11:52 /dev/vmem_diskc
brw-rw---- 1 root disk 251, 48 2010-04-18 11:52 /dev/vmem_diskd
```

其中，mkfs.ext2 /dev/vmem_diska 命令的执行会回馈如下信息：

```

root@lihacker-laptop:~/develop/svn/ldd6410-read-only/training/kernel/drivers/vmem_d
isk# mkfs.ext2 /dev/vmem_diska
mke2fs 1.41.4 (27-Jan-2009)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
64 inodes, 512 blocks
25 blocks (4.88%) reserved for the super user
First data block=1
Maximum filesystem blocks=524288
1 block group
8192 blocks per group, 8192 fragments per group
64 inodes per group

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 36 mounts or
180 days, whichever comes first. Use tune2fs -c or -i to override.

```

它将/dev/vmem_diska 格式化为 EXT2 文件系统。之后我们可以 mount 这个分区并在其中进行文件读写。

13.7.2 vmem_disk 驱动模块的加载与卸载

vmem_disk 驱动的模块加载函数完成的工作与 13.3 节给出的模板完全一致，它支持“制造请求”（对应代码清单 13.9）、请求队列（对应代码清单 13.10）两种模式（留意在请求队列方面又支持简、繁两种模式），使用模块参数 request_mode 进行区分。代码清单 13.16 给出了 vmem_disk 设备驱动的模块加载与卸载函数。

代码清单 13.17 vmem. disk 设备驱动的模块加载与卸载函数

```

1 static int __init vmem_disk_init(void)
2 {
3     int i;
4     /*
5      * 注册块设备
6      */
7     vmem_disk_major = register_blkdev(vmem_disk_major, "vmem_disk");
8     if (vmem_disk_major <= 0) {
9         printk(KERN_WARNING "vmem_disk: unable to get major number\n");
10        return -EBUSY;
11    }
12    /*
13     * 分配设备数组，初始化它们
14     */
15    Devices = kmalloc(ndevices*sizeof (struct vmem_disk_dev), GFP_KERNEL);
16    if (Devices == NULL)
17        goto out_unregister;
18    for (i = 0; i < ndevices; i++)
19        setup_device(Devices + i, i);
20
21    return 0;

```



```
22
23 out_unregister:
24 unregister_blkdev(vmem_disk_major, "sbd");
25 return -ENOMEM;
26 }
27
28 static void vmem_disk_exit(void)
29 {
30     int i;
31
32     for (i = 0; i < ndevices; i++) {
33         struct vmem_disk_dev *dev = Devices + i;
34
35         del_timer_sync(&dev->timer);
36         if (dev->gd) {
37             del_gendisk(dev->gd);
38             put_disk(dev->gd);
39         }
40         if (dev->queue) {
41             if (request_mode == RM_NOQUEUE)
42                 kobject_put(&dev->queue->kobj);
43             else
44                 blk_cleanup_queue(dev->queue);
45         }
46         if (dev->data)
47             vfree(dev->data);
48     }
49     unregister_blkdev(vmem_disk_major, "vmem_disk");
50     kfree(Devices);
51 }
52
53 /*
54  * 设置设备
55  */
56 static void setup_device(struct vmem_disk_dev *dev, int which)
57 {
58     /*
59      * 分配 globalmem 的内存
60      */
61     memset(dev, 0, sizeof(struct vmem_disk_dev));
62     dev->size = nsectors*hardsect_size;
63     dev->data = vmalloc(dev->size);
64     if (dev->data == NULL) {
65         printk(KERN_NOTICE "vmalloc failure.\n");
66         return;
67     }
68     spin_lock_init(&dev->lock);
69
70     /*
71      * 使用一个 timer 来模拟设备 invalidate
72      */
73     init_timer(&dev->timer);
74     dev->timer.data = (unsigned long) dev;
75     dev->timer.function = vmem_disk_invalidate;
76 }
```



```

77  /*
78   * I/O 队列，具体实现依赖于我们是否使用 make_request 函数
79   */
80  switch (request_mode) {
81  case RM_NOQUEUE:
82  dev->queue = blk_alloc_queue(GFP_KERNEL);
83  if (dev->queue == NULL)
84  goto out_vfree;
85  blk_queue_make_request(dev->queue, vmem_disk_make_request);
86  break;
87
88  case RM_FULL:
89  dev->queue = blk_init_queue(vmem_disk_full_request, &dev->lock);
90  if (dev->queue == NULL)
91  goto out_vfree;
92  break;
93
94  default:
95  printk(KERN_NOTICE "Bad request mode %d, using simple\n", request_mode);
96
97  case RM_SIMPLE:
98  dev->queue = blk_init_queue(vmem_disk_request, &dev->lock);
99  if (dev->queue == NULL)
100 goto out_vfree;
101 break;
102 }
103 blk_queue_hardsect_size(dev->queue, hardsect_size);
104 dev->queue->queuedata = dev;
105 /*
106  * gendisk 分配与初始化
107  */
108 dev->gd = alloc_disk(vmem_disk_MINORS);
109 if (! dev->gd) {
110 printk (KERN_NOTICE "alloc_disk failure\n");
111 goto out_vfree;
112 }
113 dev->gd->major = vmem_disk_major;
114 dev->gd->first_minor = which*vmem_disk_MINORS;
115 dev->gd->fops = &vmem_disk_ops;
116 dev->gd->queue = dev->queue;
117 dev->gd->private_data = dev;
118 snprintf (dev->gd->disk_name, 32, "vmem_disk%c", which + 'a');
119 set_capacity(dev->gd, nsectors*(hardsect_size/KERNEL_SECTOR_SIZE));
120 add_disk(dev->gd);
121 return;
122
123 out_vfree:
124 if (dev->data)
125 vfree(dev->data);
126 }

```

上述代码中引用的 `vmem_disk_major`、`ndevices`、`nsectors`、`hardsect_size` 都是模块参数，其默认值定义如下：

```

static int vmem_disk_major = 0;
module_param(vmem_disk_major, int, 0);

```



```
static int hardsect_size = 512;
module_param(hardsect_size, int, 0);
static int nsectors = 1024;    /* 该驱动器的大小 */
module_param(nsectors, int, 0);
static int ndevices = 4;
module_param(ndevices, int, 0);
/*
 * 我们能使用的不同 request 模式
 */
enum {
    RM_SIMPLE = 0, /* 简单请求函数 (对应代码清单 13.13) */
    RM_FULL = 1, /* 复杂的请求函数 (对应代码清单 13.15) */
    RM_NOQUEUE = 2, /* 使用 make_request (对应代码清单 13.16) */
};
static int request_mode = RM_SIMPLE;
module_param(request_mode, int, 0);
```

在模块加载时我们可以更改这些参数。尤其值得注意的是, `request_mode` 等于 `RM_SIMPLE`、`RM_FULL`、`RM_NOQUEUE` 分别对应于代码清单 13.13、13.15 和 13.16 这三种不同的请求处理方式。

13.7.3 vmem_disk 设备驱动 block_device_operations 及成员函数

`vmem_disk` 提供 `block_device_operations` 结构体中 `open()`、`release()`、`getgeo()`、`media_changed()`、`revalidate_disk()` 这些成员函数, 代码清单 13.18 给出了 `vmem_disk` 设备驱动的 `block_device_operations` 结构体定义及其成员函数的实现。

代码清单 13.18 vmem.disk 设备驱动 block.device.operations 结构体及成员函数

```
1 /*
2  * Open 和 close.
3  */
4
5 static int vmem_disk_open(struct block_device *bdev, fmode_t mode)
6 {
7     struct vmem_disk_dev *dev = bdev->bd_disk->private_data;
8
9     del_timer_sync(&dev->timer);
10    spin_lock(&dev->lock);
11    dev->users++;
12    spin_unlock(&dev->lock);
13
14    return 0;
15 }
16
17 static int vmem_disk_release(struct gendisk *disk, fmode_t mode)
18 {
19     struct vmem_disk_dev *dev = disk->private_data;
20
21    spin_lock(&dev->lock);
22    dev->users--;
23
24    if (!dev->users) {
25        dev->timer.expires = jiffies + INVALIDATE_DELAY;
```

```
26 add_timer(&dev->timer);
27 }
28 spin_unlock(&dev->lock);
29
30 return 0;
31 }
32
33 int vmem_disk_media_changed(struct gendisk *gd)
34 {
35     struct vmem_disk_dev *dev = gd->private_data;
36
37     return dev->media_change;
38 }
39
40 int vmem_disk_revalidate(struct gendisk *gd)
41 {
42     struct vmem_disk_dev *dev = gd->private_data;
43
44     if (dev->media_change) {
45         dev->media_change = 0;
46         memset (dev->data, 0, dev->size);
47     }
48     return 0;
49 }
50
51 /*
52  * invalidate() 在定时器到期时执行，设置一个标志来模拟磁盘的移除
53  */
54 void vmem_disk_invalidate(unsigned long ldev)
55 {
56     struct vmem_disk_dev *dev = (struct vmem_disk_dev *) ldev;
57
58     spin_lock(&dev->lock);
59     if (dev->users || !dev->data)
60         printk (KERN_WARNING "vmem_disk: timer sanity check failed\n");
61     else
62         dev->media_change = 1;
63     spin_unlock(&dev->lock);
64 }
65
66 /*
67  * getgeo() 实现
68  */
69
70 static int vmem_disk_getgeo(struct block_device *bdev, struct hd_geometry *geo)
71 {
72     long size;
73     struct vmem_disk_dev *dev = bdev->bd_disk->private_data;
74
75     size = dev->size*(hardsect_size/KERNEL_SECTOR_SIZE);
76     geo->cylinders = (size & ~0x3f) >> 6;
77     geo->heads = 4;
78     geo->sectors = 16;
79     geo->start = 4;
80 }
```



```
81 return 0;
82 }
83
84 /*
85  * block_device_operations 结构体
86  */
87 static struct block_device_operations vmem_disk_ops = {
88     .owner          = THIS_MODULE,
89     .open           = vmem_disk_open,
90     .release        = vmem_disk_release,
91     .media_changed  = vmem_disk_media_changed,
92     .revalidate_disk = vmem_disk_revalidate,
93     .getgeo         = vmem_disk_getgeo,
94 };
```

13.7.4 vmem_disk I/O 请求处理

在 vmem_disk 驱动中, 通过模块参数 request_mode 的方式来支持 3 种不同的请求处理模式以加深读者对它们的理解, 代码清单 13.19 列出了 vmem_disk 驱动的请求处理代码。

代码清单 13.19 vmem.disk 设备驱动的请求处理函数

```
1  /*
2   * 处理一个 I/O request.
3   */
4  static void vmem_disk_transfer(struct vmem_disk_dev *dev, unsigned long sector,
5  unsigned long nsect, char *buffer, int write)
6  {
7      unsigned long offset = sector*KERNEL_SECTOR_SIZE;
8      unsigned long nbytes = nsect*KERNEL_SECTOR_SIZE;
9
10     if ((offset + nbytes) > dev->size) {
11         printk (KERN_NOTICE "Beyond-end write (%ld %ld)\n", offset, nbytes);
12         return;
13     }
14     if (write)
15         memcpy(dev->data + offset, buffer, nbytes);
16     else
17         memcpy(buffer, dev->data + offset, nbytes);
18 }
19
20 /*
21  * request 函数的简单形式
22  */
23 static void vmem_disk_request(struct request_queue *q)
24 {
25     struct request *req;
26
27     while ((req = elv_next_request(q)) != NULL) {
28         struct vmem_disk_dev *dev = req->rq_disk->private_data;
29         if (! blk_fs_request(req)) {
30             printk (KERN_NOTICE "Skip non-fs request\n");
31             end_request(req, 0);
32             continue;
33         }
```

```

34
35 vmem_disk_transfer(dev, req->sector, req->current_nr_sectors,
36 req->buffer, rq_data_dir(req));
37
38 end_request(req, 1);
39 }
40 }
41
42
43 /*
44  * 传输一个单独的 BIO
45  */
46 static int vmem_disk_xfer_bio(struct vmem_disk_dev *dev, struct bio *bio)
47 {
48     int i;
49     struct bio_vec *bvec;
50     sector_t sector = bio->bi_sector;
51
52     /* Do each segment independently. */
53     bio_for_each_segment(bvec, bio, i) {
54         char *buffer = __bio_kmap_atomic(bio, i, KM_USER0);
55         vmem_disk_transfer(dev, sector, bio_cur_sectors(bio),
56         buffer, bio_data_dir(bio) == WRITE);
57         sector += bio_cur_sectors(bio);
58         __bio_kunmap_atomic(bio, KM_USER0);
59     }
60     return 0; /* Always "succeed" */
61 }
62
63 /*
64  * 传输一个完整的 request
65  */
66 static int vmem_disk_xfer_request(struct vmem_disk_dev *dev, struct request *req)
67 {
68
69     struct req_iterator iter;
70     int nsect = 0;
71     struct bio_vec *bvec;
72
73     rq_for_each_segment(bvec, req, iter) {
74         char *buffer = __bio_kmap_atomic(iter.bio, iter.i, KM_USER0);
75         sector_t sector = iter.bio->bi_sector;
76         vmem_disk_transfer(dev, sector, bio_cur_sectors(iter.bio),
77         buffer, bio_data_dir(iter.bio) == WRITE);
78         sector += bio_cur_sectors(iter.bio);
79         __bio_kunmap_atomic(iter.bio, KM_USER0);
80         nsect += iter.bio->bi_size/KERNEL_SECTOR_SIZE;
81     }
82     return nsect;
83 }
84
85
86 /*
87  * 更强大的 request 处理
88  */

```



```
89 static void vmem_disk_full_request(struct request_queue *q)
90 {
91     struct request *req;
92     int sectors_xferred;
93     struct vmem_disk_dev *dev = q->queuedata;
94
95     while ((req = elv_next_request(q)) != NULL) {
96         if (!blk_fs_request(req)) {
97             printk (KERN_NOTICE "Skip non-fs request\n");
98             end_request(req, 0);
99             continue;
100         }
101         sectors_xferred = vmem_disk_xfer_request(dev, req);
102         end_request (req, 1);
103     }
104 }
105
106 /*
107  * "制造请求"方式
108  */
109 static int vmem_disk_make_request(struct request_queue *q, struct bio *bio)
110 {
111     struct vmem_disk_dev *dev = q->queuedata;
112     int status;
113
114     status = vmem_disk_xfer_bio(dev, bio);
115     bio_endio(bio, status);
116     return 0;
117 }
```

上述代码中的 1~40 行、43~104 行、106~117 行分别对应了 RM_SIMPLE、RM_FULL、RM_NOQUEUE 这 3 种不同的请求处理方式。

vmem_disk 的驱动位于 VirtualBox 虚拟机映像的/home/lihacker/develop/svn/ldd6410-read-only/training/kernel/drivers/ vmem_disk 下面, 已经有编写好的 Makefile, 直接在其中运行 make 命令即可得到 vmem_disk.ko 模块。

13.8 实例 2: IDE 硬盘设备驱动

IDE (Integrated Drive Electronics) 接口, 也就是集成驱动器电路接口, 原名为 ATA (AT Attachment, AT 嵌入式) 接口, 其本意为将硬盘控制器与盘体集成在一起的硬盘驱动器, 经历了 ATA-1 到 ATA-7 以及 SATA-1 和 SATA-2 的发展历史。ATA-1 至 ATA-4 采用 40 芯排线, ATA-5 至 ATA-7 则采用 40 针 80 芯线缆, 虽然线缆数量增加了, 但是逻辑原理没有变, 只是通过物理上的改变来达到改善 PCB 信号完整性的目的, 它提供更多的地线并使信号线临近地线, 从而减少电流回流的面积。SATA-1 和 SATA-2 与 ATA-1 至 ATA-7 相比, 数据传输方式由并行转变为串行。

IDE 接口的硬件原理实际上非常简单, 对 CPU 的外围总线进行简单扩展后就可外接 IDE 控制器, 表 13.1 所示为 40 针 IDE 接口的引脚定义。

表 13.1 IDE 接口的引脚定义

引脚	信 号	信 号 描 述	信号方向	引脚	信 号	信 号 描 述	信号方向
1	RSET	复位	I	21	DMARQ	DMA 请求	O
2	GND	地	I/O	22	GND	地	
3	DD7	数据位 7	I/O	23	$\overline{\text{IDOW}}$	写选通	I
4	DD8	数据位 8	I/O	24	GND	地	
5	DD6	数据位 6	I/O	25	$\overline{\text{DIOR}}$	读选通	I
6	DD9	数据位 9	I/O	26	GND	地	
7	DD5	数据位 5	I/O	27	IORDY	通道就绪	O
8	DD10	数据位 10	I/O	28	DPSYNC: CXEL	同步电缆选择	
9	DD4	数据位 4	I/O	29	$\overline{\text{DMACK}}$	DMA 应答	O
10	DD11	数据位 11	I/O	30	GND	地	
11	DD3	数据位 3	I/O	31	$\overline{\text{INTRQ}}$	中断请求	O
12	DD12	数据位 12	I/O	32	$\overline{\text{IOCS16}}$	16 位 I/O 片选	O
13	DD2	数据位 2	I/O	33	DA1	地址 1	I
14	DD13	数据位 13	I/O	34	$\overline{\text{PDIAG}}$	诊断完成	O
15	DD1	数据位 1	I/O	35	DA0	地址 0	I
16	DD14	数据位 14	I/O	36	DA2	地址 2	I
17	DD0	数据位 0	I/O	37	$\overline{\text{CS0}}$	片选 0	I
18	DD15	数据位 15	I/O	38	$\overline{\text{CS1}}$	片选 1	I
19	GND	地		39	$\overline{\text{DASP}}$	驱动器状态指示	O
20	N.C	未用		40	GND	地	

IDE 控制器提供了一组寄存器，通过这些寄存器，主机能控制 IDE 驱动器的行为和查询其状态，表 13.2 所示 IDE 接口寄存器的定义。

表 13.2 IDE 接口寄存器定义

片选 1	片选 0	地址 2	地址 1	地址 0	读	写	位数
1	0	0	0	0	数据寄存器	数据寄存器	16
1	0	0	0	1	错误寄存器	特征寄存器	8
1	0	0	1	0	扇区数寄存器	扇区数寄存器	8
1	0	0	1	1	扇区号寄存器	扇区号寄存器	8
1	0	1	0	0	柱面号寄存器（低 8 位）	柱面号寄存器（低 8 位）	8
1	0	1	0	1	柱面号寄存器（高 8 位）	柱面号寄存器（高 8 位）	8
1	0	1	1	0	驱动器选择/磁头寄存器	驱动器选择/磁头寄存器	8
1	0	1	1	1	状态寄存器	命令寄存器	8
0	1	1	1	0	状态寄存器	设备控制器寄存器	8

IDE 硬盘的传输模式有以下 3 种。



- **PIO (Programmed I/O) 模式:** PIO 模式是一种通过 CPU 执行 I/O 端口指令来进行数据读写的交换模式, 是最早的硬盘数据传输模式, 数据传输速率低下, CPU 占有率也很高。
- **DMA (Direct Memory Access) 模式:** DMA 模式是一种不经过 CPU 而直接从内存存取数据的数据交换模式。PIO 模式下硬盘和内存之间的数据传输是由 CPU 来控制的; 而在 DMA 模式下, CPU 只需向 DMA 控制器下达指令, 让 DMA 控制器来处理数据的传送, 数据传送完毕再把信息反馈给 CPU, 这样就在很大程度上减轻了 CPU 资源占有率。
- **Ultra DMA (简称 UDMA) 模式:** 它在包含了 DMA 模式的优点的基础上又增加了 CRC 校验技术, 提高数据传输过程中的准确性, 安全性得到保障。另外, 在以往的硬盘数据传输模式下, 一个时钟周期只传输一次数据, 而在 UDMA 模式中逐渐应用了 Double Data Rate (双倍数据传输) 技术, 它在时钟的上升沿和下降沿各自进行一次数据传输, 使数据传输速度成倍增长。

除了可以以 CHS (Cylinder、Head 和 Sector) 的方式定位硬盘的扇区外, 还可以用 LBA (逻辑块线性地址) 的方式来定位, CHS 可以换算为 LBA。CHS 设计最多只允许 65536 个柱面、16 个磁头以及 255 个扇区/磁轨。这就将容量限制为 267386880 个扇区, 即大约 137GB。

假设用 c 表示当前柱面号, h 表示当前磁头号, cs 表示起始柱面号, hs 表示起始磁头号, ss 表示起始扇区号, ps 表示每磁道有多少个扇区, ph 表示每柱面有多少个磁道 (一般情况下, $cs = 0$ 、 $hs = 0$ 、 $ss = 1$ 、 $ps = 63$ 、 $ph = 255$), LBA 与 CHS 有如下对应关系:

$$lba = (c - cs) * ph * ps + (h - hs) * ps + (s - ss)$$

LBA 使得系统忽略硬盘的几何结构, 交由驱动器来完成。系统不需要去查询 CHS 值, 而只需要查询逻辑块地址 (Logical Block Address, LBA), 驱动器电子装置会找出要读或写的实际扇区。而 LBA48 (48 位逻辑块地址) 则可以使系统支持超过 137GB 的硬盘。

Linux 内核中, 早期常使用 `drivers/ide` 目录下的驱动, 而如今在嵌入式系统中则多使用 `drivers/ata` 下的驱动, 尤其值得一提的是 `drivers/ata` 的 `pata_platform.c` 和 `pata_*.c` 文件联合起来实现了一个平台无关的并行 ATA 驱动, 这种情况下, 连接了嵌入式硬盘的电路板只需要在板文件中添加与并行 ATA 设备相关的平台设备和资源信息即可, 不需要编写一行驱动代码就可以让硬盘工作。代码清单 13.20 给出了 `arch/arm/mach-rpc/riscpc.c` 板文件中新增 `pata_platform` 设备的例子。

代码清单 13.20 使用 pata_platform 驱动连接 IDE 硬盘

```
1 static struct pata_platform_info pata_platform_data = {
2     .ioport_shift    = 2,
3 };
4
5 static struct resource pata_resources[] = {
6     [0] = {
7         .start        = 0x030107c0,
8         .end          = 0x030107df,
9         .flags        = IORESOURCE_MEM,
10    },
11    [1] = {
12        .start        = 0x03010fd8,
13        .end          = 0x03010fdb,
14        .flags        = IORESOURCE_MEM,
```



```

15     },
16     [2] = {
17         .start      = IRQ_HARDDISK,
18         .end        = IRQ_HARDDISK,
19         .flags      = IORESOURCE_IRQ,
20     },
21 };
22
23 static struct platform_device pata_device = {
24     .name          = "pata_platform",
25     .id            = -1,
26     .num_resources = ARRAY_SIZE(pata_resources),
27     .resource       = pata_resources,
28     .dev           = {
29         .platform_data = &pata_platform_data,
30         .coherent_dma_mask = ~0,          /* grumble */
31     },
32 };

```

上述代码中的两个 `IORESOURCE_MEM` 资源对应于 IDE 硬盘连接的两个内存区域。
`IORESOURCE_IRQ` 资源是硬盘使用的 CPU 的中断号。

13.9 总结

块设备的 I/O 操作方式与字符设备存在较大的不同，因而引入了 `request_queue`、`request`、`bio` 等一系列数据结构。在整个块设备的 I/O 操作中，贯穿于始终的就是“请求”，字符设备的 I/O 操作则是直接进行不绕弯，块设备的 I/O 操作会排队和整合。

驱动的任务是处理请求，对请求的排队和整合由 I/O 调度算法解决，因此，块设备驱动的核心就是请求处理函数或“制造请求”函数。

尽管在块设备驱动中仍然存在 `block_device_operations` 结构体及其成员函数，但其不再包含读写一类的成员函数，而只是包含打开、释放及 I/O 控制等与具体读写无关的函数。

块设备驱动的结构相当复杂的，但幸运的是，块设备不像字符设备那么包罗万象，它通常就是存储设备，而且驱动的主体已经由 Linux 内核提供，针对一个特定的硬件系统，驱动工程师所涉及的工作往往只是编写极其少量的与硬件平台相关的代码。

LINUX

第14章 Linux 终端设备驱动

本章导读

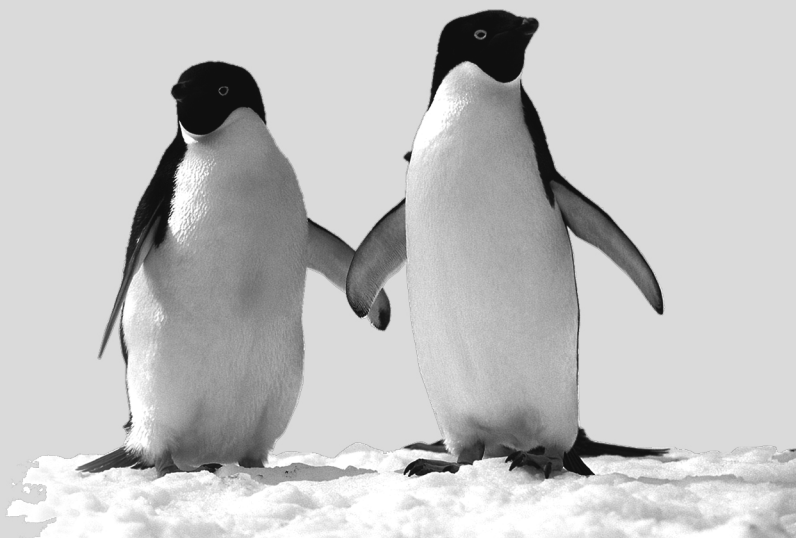
在 Linux 系统中，终端设备非常重要，没有终端设备，系统将无法向用户反馈信息，Linux 中包含控制台、串口和伪终端 3 类终端设备。

14.1 节讲解了终端设备的概念及分类，14.2 节讲解了 Linux 终端设备驱动的框架结构，重点描述 `tty_driver` 结构体及其成员。

14.3~14.5 节在 14.2 节的基础上，分别讲解了 Linux 终端设备驱动模块加载/卸载函数和 `open()`、`close()` 函数，数据读写流程及 `tty` 设备线路设置的编程方法。

在 Linux 中，串口驱动完全遵循 `tty` 驱动的框架结构，但是进行了底层操作的再次封装，14.6 节描述了 Linux 针对串口 `tty` 驱动的这一封装，14.7 节则具体给出了串口 `tty` 驱动的实现方法。

14.8 节基于 14.6 和 14.7 节的讲解给出了串口 `tty` 驱动的设计实例，即 S3C6410 集成 UART 的驱动。



14.1 终端设备

在 Linux 系统中，终端是一种字符型设备，它有多种类型，通常使用 `tty` 来简称各种类型的终端设备。`tty` 是 Teletype 的缩写，Teletype 是最早出现的一种终端设备，很像电传打字机，是由 Teletype 公司生产的。Linux 中包含如下几类终端设备。

1. 串行端口终端 (/dev/ttySn)

串行端口终端 (Serial Port Terminal) 是使用计算机串行端口连接的终端设备。计算机把每个串行端口都看作是一个字符设备。这些串行端口所对应的设备名称是 `/dev/ttyS0` (或 `/dev/tts/0`)、`/dev/ttyS1` (或 `/dev/tts/1`) 等，设备号分别是 (4, 0)、(4, 1) 等。

在命令行上把标准输出重定向到端口对应的设备文件名上就可以通过该端口发送数据，例如，在命令行提示符下键入：`echo test > /dev/ttyS1` 会把单词 “test” 发送到连接在 `ttyS1` 端口的设备上。

目前 USB-串口转换器也已经非常常用，其对应设备结点通常为 `/dev/ttyUSB0`、`/dev/ttyUSB1` 等。

2. 伪终端 (/dev/pty/)

伪终端 `pty` (Pseudo Terminal) 是成对的逻辑终端设备，并存在成对的设备文件，如 `/dev/ptyp3` 和 `/dev/ttyp3`，它们与实际物理设备并不直接相关。如果一个程序把 `ttyp3` 看作是一个串行端口设备，则它对该端口的读/写操作会反映在该逻辑终端设备对应的 `ttyp3` 上，而 `ttyp3` 则是另一个程序用于读写操作的逻辑设备。这样，两个程序就可以通过这种逻辑设备进行互相交流，使用 `ttyp3` 的程序会认为自己正在与一个串行端口进行通信。

以 `telnet` 为例，如果某人使用 `telnet` 程序连接到 Linux 系统，则 `telnet` 程序就可能开始连接到设备 `ptyp2` 上，而此时一个 `getty` 程序会运行在对应的 `ttyp2` 端口上。当 `telnet` 从远端获取了一个字符时，该字符就会通过 `ptyp2`、`ttyp2` 传递给 `getty` 程序，而 `getty` 程序则会通过 `ttyp2`、`ptyp2` 和 `telnet` 程序返回 “login:” 字符串信息。这样，登录程序与 `telnet` 程序就通过伪终端进行通信。通过使用适当的软件，可以把两个或多个伪终端设备连接到同一个物理串行端口上。

而目前的 Linux 版本常采用 `pts` (pseudo-terminal slave) 与 `ptmx` (pseudo-terminal master) 配合的方法来实现 `pty`。目录 `/dev/pts` 是一个类型为 `devpts` 的文件系统，并且可以在被加载文件系统列表中看到。`/dev/ptmx` 是 1 个主设备号为 5，次设备号为 2 的字符设备，它被用于创建一个 master/slave 对。当某进程打开 `/dev/ptmx` 的时候，它将得到一个 master 的文件描述符，每个被打开的文件描述符对应一个独立的 master，而且对应一个 `pts`，将该文件描述符作为参数传入 `ptsname` 可以得到 `pts` 的路径。

3. 控制终端 (/dev/tty)

如果当前进程有控制终端 (Controlling Terminal) 的话，那么 `/dev/tty` 就是当前进程的控制终端的设备特殊文件。可以使用命令 “`ps -ax`” 来查看进程与哪个控制终端相连，使用命令 “`tty`” 可以查看它具体对应哪个实际终端设备。`/dev/tty` 有些类似于到实际所使用终端设备的一个 link，例如：

```
lihacker@lihacker-laptop:/$ ps ax
PID TTY      STAT   TIME COMMAND
```



```
1 ?      Ss      0:06 /sbin/init
2 ?      S<      0:00 [kthreadd]
3 ?      S<      0:00 [migration/0]
...
783 ?    S<S     0:02 /sbin/udev --daemon
995 ?    S<      0:00 [kpsmoused]
...
2012 tty4 Ss+     0:00 /sbin/getty 38400 tty4
2013 tty5 Ss+     0:00 /sbin/getty 38400 tty5
2019 tty2 Ss+     0:00 /sbin/getty 38400 tty2
2021 tty3 Ss+     0:00 /sbin/getty 38400 tty3
2022 tty6 Ss+     0:00 /sbin/getty 38400 tty6
2093 ?    Ss      0:00 /usr/sbin/acpid -c /etc/acpi/events -s
/var/run/acpid.socket
...
4075 pts/0 Ss      0:03 bash
4580 pts/1 Ss      0:01 bash
4596 pts/1 S+      0:00 ssh lihacker@10.0.2.15
4597 ?    Ss      0:00 sshd: lihacker [priv]
4606 ?    S       0:00 sshd: lihacker@pts/2
4608 pts/2 Ss+     0:01 -bash
4952 pts/0 R+      0:00 ps ax
```

内核线程（如 `kthreadd`）和用户空间的守护进程（如 `udev`）是没有控制终端的，因此其 `tty` 栏目标注的是“?”。

4. 控制台终端（`/dev/tty`，`/dev/console`）

在 UNIX 系统中，计算机显示器通常被称为控制台终端（`console`）。它仿真了类型为 Linux 的一种终端（`TERM=Linux`），并且有一些设备特殊文件与之相关联：`tty0`、`tty1`、`tty2` 等。当用户在控制台上登录时，使用的是 `tty1`。使用 `Alt+[F1~F6]` 组合键时，我们就可以切换到 `tty2`、`tty3` 等上面去。`tty1~tty6` 等称为虚拟终端，而 `tty0` 则是当前所使用虚拟终端的一个别名，系统所产生的信息会发送到该终端上。因此不管当前正在使用哪个虚拟终端，系统信息都会发送到控制台终端上。用户可以登录到不同的虚拟终端上去，因而可以让系统同时有几个不同的会话期存在。只有系统或超级用户 `root` 可以向 `/dev/tty0` 进行写操作。

在 Linux 中，可以在系统启动命令行里指定当前的输出终端，格式如下：

```
console=device, options
```

`device` 指代的是终端设备，可以是 `tty0`（前台的虚拟终端）、`ttyX`（第 `X` 个虚拟终端）、`ttySX`（第 `X` 个串口）、`lp0`（第一个并口）等。

`options` 指代对 `device` 进行的设置，它取决于具体的设备驱动。对于串口设备，参数用来定义为：

波特率、校验位、位数，格式为 `BBBBPN`，其中 `BBBB` 表示波特率，`P` 表示校验（`n/o/e`），`N` 表示位数，默认 `options` 是 `9600n8`。

用户可以在内核命令行中同时设定多个 `console`，这样输出将会在所有的 `console` 上显示，而当用户调用 `open()` 打开 `/dev/console` 时，最后一个 `console` 将会返回作为当前值。例如：

```
console=ttyS1, 9600 console=tty0
```

定义了两个 `console`，而调用 `open()` 打开 `/dev/console` 时，将使用虚拟终端 `tty0`。但是内核消息会在 `tty0` VGA 虚拟终端和串口 `ttyS1` 上同时显示。简单地说，我们可以把 `/dev/console` 看

作内核控制台的 `tty` 文件接口，设备号位 `0x0501`，当对其调用 `tty_open()`时，它会转义为实际的终端设备。

通过查看 `/proc/tty/drivers` 文件可以获知什么类型的 `tty` 设备存在以及什么驱动被加载到内核，这个文件包括一个当前存在的不同 `tty` 驱动的列表，包括驱动名、缺省的节点名、驱动的主编号、这个驱动使用的次编号范围，以及 `tty` 驱动的类型。例如，下面给出了一个 `/proc/tty/drivers` 文件的例子：

```
[root@localhost root]# cat /proc/tty/drivers
/dev/tty          /dev/tty          5          0 system:/dev/tty
/dev/console      /dev/console      5          1 system:console
/dev/ptmx         /dev/ptmx         5          2 system
/dev/vc/0         /dev/vc/0         4          0 system:vtmaster
serial           /dev/ttys         4 64-67 serial
pty_slave        /dev/pts          136 0-1048575 pty:slave
pty_master       /dev/ptm          128 0-1048575 pty:master
pty_slave        /dev/ttyp         3 0-255 pty:slave
pty_master       /dev/pty          2 0-255 pty:master
unknown          /dev/tty          4 1-63 console
```

14.2 终端设备驱动结构

Linux 内核中 `tty` 的层次结构如图 14.1 所示，包含 `tty` 核心、`tty` 线路规程和 `tty` 驱动，`tty` 线路规程的工作是以特殊的方式格式化从一个用户或者硬件收到的数据，这种格式化常常采用一个协议转换的形式，例如 `PPP` 和 `Bluetooth`。`tty` 设备发送数据的流程为：`tty` 核心从一个用户获取将要发送的一个 `tty` 设备的数据，`tty` 核心将数据传递给 `tty` 线路规程驱动，接着数据被传递到 `tty` 驱动，`tty` 驱动将数据转换为可以发送给硬件的格式。接收数据的流程为：从 `tty` 硬件接收到的数据向上交给 `tty` 驱动，进入 `tty` 线路规程驱动，再进入 `tty` 核心，在这里它被一个用户获取。尽管大多数时候 `tty` 核心和 `tty` 之间的数据传输会经历 `tty` 线路规程的转换，但是 `tty` 驱动与 `tty` 核心之间也可以直接传输数据。

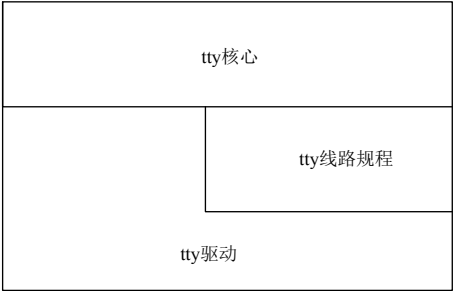


图 14.1 tty 分层结构

图 14.2 显示了与 `tty` 相关的主要源文件及数据的流向。`drivers/char/tty_io.c` 定义了 `tty` 设备通用的 `file_operations` 结构体并实现了接口函数 `tty_register_driver()`用于注册 `tty` 设备，它会利用 `fs/char_dev.c` 提供的接口函数注册字符设备，与具体设备对应的 `tty` 驱动将实现 `tty_driver` 结构体中的成员函数。同时 `tty_io.c` 也提供了 `tty_register_ldisc()`接口函数用于注册线路规程，典型地，例如 `drivers/char/n_tty.c` 文件则针对 `N_TTY` 线路规程实现了 `tty_disc` 结构体中的成员。

从图 14.2 可以看出，特定 `tty` 设备驱动的主体工作是填充 `tty_driver` 结构体中的成员，实现其中的成员函数，`tty_driver` 结构体的定义如代码清单 14.1。

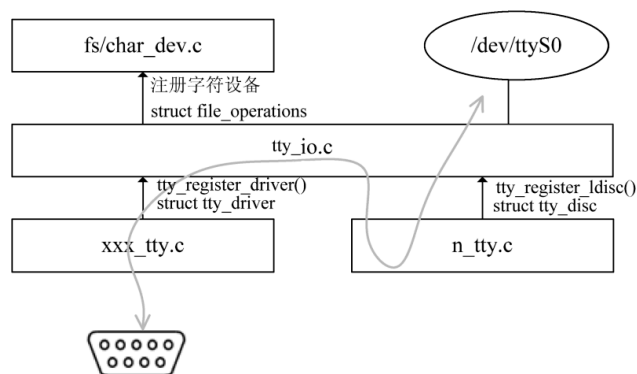


图 14.2 tty 主要源文件关系及数据流向

代码清单 14.1 tty_driver 结构体

```
1 struct tty_driver {
2     int magic; /* 该结构体的幻数 */
3     struct kref kref; /* Reference 管理 */
4     struct cdev cdev;
5     struct module *owner;
6     const char *driver_name;
7     const char *name;
8     int name_base;
9     int major; /* 主设备号 */
10    int minor_start; /* 开始次设备号 */
11    int minor_num; /* 可能的设备数量 */
12    int num; /* 被分配设备的数量 */
13    short type; /* tty 驱动的类型 */
14    short subtype; /* tty 驱动的子类 */
15    struct ktermios init_termios; /* 初始的 termios */
16    int flags; /* tty 驱动标志 */
17    struct proc_dir_entry *proc_entry; /* /proc 入口 */
18    struct tty_driver *other; /* 仅对 PTY 驱动有用 */
19
20    /*
21     * tty 数据结构指针
22     */
23    struct tty_struct **ttys;
24    struct ktermios **termios;
25    struct ktermios **termios_locked;
26    void *driver_state;
27
28    /*
29     * 驱动中的操作函数
30     */
31
32    const struct tty_operations *ops;
33    struct list_head tty_drivers;
34 };
```

tty_driver 结构体中的 magic 表示给这个结构体的“幻数”，设为 TTY_DRIVER_MAGIC（即 0x5402），在 alloc_tty_driver() 函数中被初始化。

name 与 driver_name 的不同在于后者表示驱动的名字，用在 /proc/tty 和 sysfs 中，而前者表

示驱动的设备节点名。

`type` 与 `subtype` 描述 `tty` 驱动的类型和子类型, `subtype` 的值依赖于 `type`, `type` 成员的可能值为 `TTY_DRIVER_TYPE_SYSTEM` (`subtype` 应当设为 `SYSTEM_TYPE_TTY`、`SYSTEM_TYPE_CONSOLE`、`SYSTEM_TYPE_SYSCONS` 或 `SYSTEM_TYPE_SYSPTMX`)、`TTY_DRIVER_TYPE_CONSOLE` (仅被控制台驱动使用)、`TTY_DRIVER_TYPE_SERIAL` (被任何串行类型驱动使用, `subtype` 通常设为 `SERIAL_TYPE_NORMAL`)、`TTY_DRIVER_TYPE_PTY` (被伪控制台接口 `pty` 使用, 此时 `subtype` 需要被设置为 `PTY_TYPE_MASTER` 或 `PTY_TYPE_SLAVE`)、`TTY_DRIVER_TYPE_SCC` (由 `SCC` 驱动使用)。

`init_termios` 为初始线路设置, 为一个 `termios` 结构体, 这个成员被用来提供一个线路设置集合。

`termios` 用于保存当前的线路设置, 这些线路设置控制当前波特率、数据大小、数据流控设置等, 这个结构体包含 `tcflag_t c_iflag` (输入模式标志)、`tcflag_t c_oflag` (输出模式标志)、`tcflag_t c_cflag` (控制模式标志)、`tcflag_t c_lflag` (本地模式标志)、`cc_t c_line` (线路规程类型)、`cc_t c_cc[NCCS]` (一个控制字符数组) 等成员。

驱动会使用一个标准的数值集初始化这个成员, 它拷贝自 `tty_std_termios` 变量, `tty_std_termios` 在 `tty` 核心中的定义如代码清单 14.2。

代码清单 14.2 `tty_std_termios` 变量

```
1 struct ktermios tty_std_termios = {
2     .c_iflag = ICRNL | IXON, /* 输入模式 */
3     .c_oflag = OPOST | ONLCR, /* 输出模式 */
4     .c_cflag = B38400 | CS8 | CREAD | HUPCL, /* 控制模式 */
5     .c_lflag = ISIG | ICANON | ECHO | ECHOE | ECHOK |
6         ECHOCTL | ECHOKE | IEXTEN, /* 本地模式 */
7     .c_cc = INIT_C_CC, /* 控制字符, 用来修改终端的特殊字符映射 */
8     .c_ispeed = 38400,
9     .c_ospeed = 38400
10 };
```

`tty_driver` 结构体中的 `major`、`minor_start`、`minor_num` 表示主设备号、次设备号及可能的次设备数, `name` 表示设备名 (如 `ttyS`), 第 32 行是一个 `tty_operations` 结构体的指针, 它的定义如代码清单 14.3, 其成员函数通常需在特定设备 `tty` 驱动模块初始化函数中被赋值。

代码清单 14.3 `tty_operations` 结构体

```
1 struct tty_operations {
2     struct tty_struct * (*lookup)(struct tty_driver *driver,
3     struct inode *inode, int idx);
4     int (*install)(struct tty_driver *driver, struct tty_struct *tty);
5     void (*remove)(struct tty_driver *driver, struct tty_struct *tty);
6     int (*open)(struct tty_struct *tty, struct file * filp);
7     void (*close)(struct tty_struct *tty, struct file * filp);
8     void (*shutdown)(struct tty_struct *tty);
9     int (*write)(struct tty_struct *tty,
10     const unsigned char *buf, int count);
11     int (*put_char)(struct tty_struct *tty, unsigned char ch);
12     void (*flush_chars)(struct tty_struct *tty);
13     int (*write_room)(struct tty_struct *tty);
14     int (*chars_in_buffer)(struct tty_struct *tty);
15     int (*ioctl)(struct tty_struct *tty, struct file * file,
16     unsigned int cmd, unsigned long arg);
```



```
17 long (*compat_ioctl)(struct tty_struct *tty, struct file * file,  
18                      unsigned int cmd, unsigned long arg);  
19 void (*set_termios)(struct tty_struct *tty, struct ktermios * old);  
20 void (*throttle)(struct tty_struct * tty);  
21 void (*unthrottle)(struct tty_struct * tty);  
22 void (*stop)(struct tty_struct *tty);  
23 void (*start)(struct tty_struct *tty);  
24 void (*hangup)(struct tty_struct *tty);  
25 int (*break_ctl)(struct tty_struct *tty, int state);  
26 void (*flush_buffer)(struct tty_struct *tty);  
27 void (*set_ldisc)(struct tty_struct *tty);  
28 void (*wait_until_sent)(struct tty_struct *tty, int timeout);  
29 void (*send_xchar)(struct tty_struct *tty, char ch);  
30 int (*read_proc)(char *page, char **start, off_t off,  
31                 int count, int *eof, void *data);  
32 int (*tiocmget)(struct tty_struct *tty, struct file *file);  
33 int (*tiocmset)(struct tty_struct *tty, struct file *file,  
34                unsigned int set, unsigned int clear);  
35 int (*resize)(struct tty_struct *tty, struct tty_struct *real_tty,  
36               struct winsize *ws);  
37 int (*set_termiox)(struct tty_struct *tty, struct termiox *tnew);  
38 #ifdef CONFIG_CONSOLE_POLL  
39 int (*poll_init)(struct tty_driver *driver, int line, char *options);  
40 int (*poll_get_char)(struct tty_driver *driver, int line);  
41 void (*poll_put_char)(struct tty_driver *driver, int line, char ch);  
42 #endif  
43};
```

put_char()为单字节写函数,当单个字节被写入设备时这个函数被 tty 核心调用,如果一个 tty 驱动没有定义这个函数,将使用 count 参数为 1 的 write()函数。

flush_chars()与 wait_until_sent()函数都用于刷新数据到硬件。

write_room()指示有多少缓冲区空闲,chars_in_buffer()指示缓冲区中包含的数据数。

当在 tty 设备的设备节点上执行 IOCTL 操作时, tty_operations 结构体的 ioctl()函数会被 tty 核心调用。

当设备的 termios 设置被改变时, set_termios()函数将被 tty 核心调用。

throttle()、unthrottle()、stop()和 start()为数据抑制函数,这些函数用来辅助控制 tty 核心的输入缓冲区。当 tty 核心的输入缓冲区满时, throttle()函数将被调用, tty 驱动试图通知设备不应当发送字符给它。当 tty 核心的输入缓冲区已被清空时, unthrottle()函数将被调用以暗示设备可以接收数据。stop()和 start()函数非常像 throttle()和 unthrottle()函数,但它们表示 tty 驱动应当停止发送数据给设备以及恢复发送数据。

当 tty 驱动挂起 tty 设备时, hangup()函数被调用,在此函数中进行相关的硬件操作。

当 tty 驱动要在 RS-232 端口上打开或关闭线路的 BREAK 状态时, break_ctl()线路中断控制函数被调用。如果 state 状态设为-1, BREAK 状态打开,如果状态设为 0, BREAK 状态关闭。如果这个函数由 tty 驱动实现,而 tty 核心将处理 TCSBRK、TCSBRKP、TIOCSBRK 和 TIOCCBRK 这些 IOCTL 命令。

flush_buffer()函数用于刷新缓冲区并丢弃任何剩下的数据。

set_ldisc()函数用于设置线路规程,当 tty 核心改变 tty 驱动的线路规程时这个函数被调用,这个函数通常不需要被驱动定义。

send_xchar()为 X-类型字符发送函数,这个函数用来发送一个高优先级 XON 或者 XOFF 字符

给 tty 设备，要被发送的字符在第 2 个参数 ch 中指定。

read_proc()和 write_proc()为/proc 读和写函数。

tiocmget()函数用于获得 tty 设备的线路设置，对应的 tiocmset()用于设置 tty 设备的线路设置，参数 set 和 clear 包含了要设置或者清除的线路设置。

Linux 内核提供了一组函数用于操作 tty_driver 结构体及 tty 设备，如下所示。

(1) 分配 tty 驱动。

```
struct tty_driver *alloc_tty_driver(int lines);
```

这个函数返回 tty_driver 指针，其参数为要分配的设备数量，line 会被赋值给 tty_driver 的 num 成员。

(2) 注册 tty 驱动。

```
int tty_register_driver(struct tty_driver *driver);
```

参数为由 alloc_tty_driver ()分配的 tty_driver 结构体指针，注册 tty 驱动成功时返回 0。

(3) 注销 tty 驱动。

```
int tty_unregister_driver(struct tty_driver *driver);
```

这个函数与 tty_register_driver ()对应，tty 驱动最终会调用上述函数注销 tty_driver。

(4) 注册 tty 设备。

```
void tty_register_device(struct tty_driver *driver, unsigned index,
                        struct device *device);
```

仅有 tty_driver 是不够的，驱动必须依附于设备，tty_register_device()函数用于注册关联于 tty_driver 的设备，index 为设备的索引（范围是 0~driver->num），如：

```
for (i = 0; i < XXX_TTY_MINORS; ++i)
    tty_register_device(xxx_tty_driver, i, NULL);
```

(5) 注销 tty 设备。

```
void tty_unregister_device(struct tty_driver *driver, unsigned index);
```

上述函数与 tty_register_device()对应，用于注销 tty 设备，其使用方法如：

```
for (i = 0; i < XXX_TTY_MINORS; ++i)
    tty_unregister_device(xxx_tty_driver, i);
```

(6) 设置 tty 驱动操作。

```
void tty_set_operations(struct tty_driver *driver, struct tty_operations *op);
```

上述函数会将 tty_operations 结构体中的函数指针拷贝给 tty_driver 对应的函数指针。

终端设备驱动都围绕 tty_driver 结构体而展开，一般而言，终端设备驱动应包含如下组成。

(1) 终端设备驱动模块加载函数和卸载函数，完成注册和注销 tty_driver，初始化和释放终端设备对应的 tty_driver 结构体成员及硬件资源。

(2) 实现 tty_operations 结构体中的一系列成员函数，主要是实现 open()、close()、write()、tiocmget()、tiocmset()等函数。

14.3 终端设备驱动初始化与释放

14.3.1 模块加载与卸载函数

tty 驱动模块的加载函数中通常需要分配、初始化 tty_driver 结构体并申请必要的硬件资源，



如代码清单 14.4。tty 驱动的设备卸载函数完成与模块加载函数相反的工作。

代码清单 14.4 终端设备驱动模块加载函数范例

```
1 /* tty 驱动模块加载函数 */
2 static int __init xxx_init(void)
3 {
4     ...
5     /* 分配 tty_driver 结构体 */
6     xxx_tty_driver = alloc_tty_driver(XXX_PORTS);
7     /* 初始化 tty_driver 结构体 */
8     xxx_tty_driver->owner = THIS_MODULE;
9     xxx_tty_driver->name = "ttyS";
10    xxx_tty_driver->major = TTY_MAJOR;
11    xxx_tty_driver->minor_start = 64;
12    xxx_tty_driver->type = TTY_DRIVER_TYPE_SERIAL;
13    xxx_tty_driver->subtype = SERIAL_TYPE_NORMAL;
14    xxx_tty_driver->init_termios = tty_std_termios;
15    xxx_tty_driver->init_termios.c_cflag = B9600 | CS8 | CREAD | HUPCL | CLOCAL;
16    xxx_tty_driver->flags = TTY_DRIVER_REAL_RAW;
17    tty_set_operations(xxx_tty_driver, &xxx_ops);
18
19    ret = tty_register_driver(xxx_tty_driver);
20    if (ret) {
21        printk(KERN_ERR "Couldn't register xxx serial driver\n");
22        put_tty_driver(xxx_tty_driver);
23        return ret;
24    }
25
26    ...
27    ret = request_irq(...); /* 硬件资源申请 */
28    ...
29 }
```

14.3.2 打开与关闭函数

当用户对 tty 驱动所分配的设备节点进行 `open()` 系统调用时, `tty_driver` 所拥有的 `tty_operations` 中的 `open()` 成员函数将被 tty 核心调用。tty 驱动必须设置 `open()` 成员, 否则, `-ENODEV` 将被返回给调用 `open()` 的用户。

`open()` 成员函数的第 1 个参数为一个指向分配给这个设备的 `tty_struct` 结构体的指针, 第 2 个参数为文件指针。

`tty_struct` 结构体被 tty 核心用来保存当前 tty 端口的状态, 它的大多数成员只被 tty 核心使用。`tty_struct` 中的几个重要成员如下。

(1) `flags` 标示 tty 设备的当前状态, 包括 `TTY_THROTTLED`、`TTY_IO_ERROR`、`TTY_OTHER_CLOSED`、`TTY_EXCLUSIVE`、`TTY_DEBUG`、`TTY_DO_WRITE_WAKEUP`、`TTY_PUSH`、`TTY_CLOSING`、`TTY_DONT_FLIP`、`TTY_HW_COOK_OUT`、`TTY_HW_COOK_IN`、`TTY_PTY_LOCK`、`TTY_NO_WRITE_SPLIT` 等。

(2) `ldisc` 为给 tty 设备的线路规程。

(3) `write_wait`、`read_wait` 为给 tty 写/读函数的等待队列, tty 驱动应当在合适的时机唤醒对

应的等待队列。

(4) `termios` 为指向 `tty` 设备的当前 `termios` 设置的指针。

(5) `stopped:1` 指示是否停止 `tty` 设备, `tty` 驱动可以设置这个值; `hw_stopped:1` 指示是否 `tty` 设备已经被停止, `tty` 驱动可以设置这个值; `flow_stopped:1` 指示是否 `tty` 设备数据流停止。

(6) `driver_data`、`disc_data` 为数据指针, 用于存储 `tty` 驱动和线路规程的“私有”数据。

驱动中可以定义一个设备相关的结构体, 并在 `open()` 函数中将其赋值给 `tty_struct` 的 `driver_data` 成员, 如代码清单 14.5。

代码清单 14.5 在 `tty` 驱动打开函数中赋值 `tty_struct` 的 `driver_data` 成员

```

1  /* 设备“私有”数据结构体 */
2  struct xxx_tty {
3      struct tty_struct *tty; /* tty_struct 指针 */
4      int open_count; /* 打开次数 */
5      struct semaphore sem; /* 结构体锁定信号量 */
6      int xmit_buf; /* 传输缓冲区 */
7      ...
8  }
9
10 /* 打开函数 */
11 static int xxx_open(struct tty_struct *tty, struct file *file)
12 {
13     struct xxx_tty *xxx;
14
15     /* 分配 xxx_tty */
16     xxx = kmalloc(sizeof(*xxx), GFP_KERNEL);
17     if (!xxx)
18         return -ENOMEM;
19     /* 初始化 xxx_tty 中的成员 */
20     init_MUTEX(&xxx->sem);
21     xxx->open_count = 0;
22     ...
23     /* 让 tty_struct 中的 driver_data 指向 xxx_tty */
24     tty->driver_data = xxx;
25     xxx->tty = tty;
26     ...
27     return 0;
28 }

```

在用户对前面使用 `open()` 系统调用而创建的文件句柄进行 `close()` 系统调用时, `tty_driver` 所拥有的 `tty_operations` 中的 `close()` 成员函数将被 `tty` 核心调用。

14.4 数据发送和接收

图 14.3 所示终端设备数据发送和接收过程中的数据流以及函数调用关系。用户在有数据发送给终端设备时, 通过“`write()` 系统调用——`tty` 核心——线路规程”的层层调用, 最终调用 `tty_driver` 结构体中的 `write()` 函数完成发送。

因为传输速度和 `tty` 硬件缓冲区容量的原因, 不是所有的写程序要求的字符都可以在调用写



函数时被发送, 因此写函数应当返回能够发送给硬件的字节数以使用户程序检查是否所有的数据被真正写入。如果在 `write()` 调用期间发生任何错误, 一个负的错误码应当被返回。

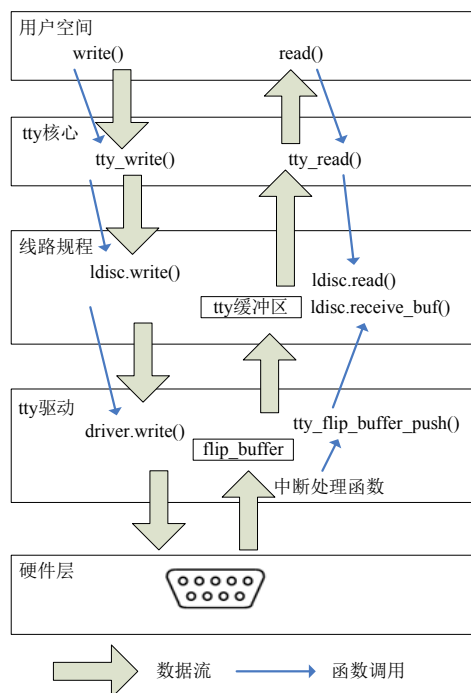


图 14.3 终端设备数据发送和接收过程中的数据流和函数调用关系

`tty_driver` 的 `write()` 函数接受 3 个参数 `tty_struct`、发送数据指针及要发送的字节数, 一般首先会通过 `tty_struct` 的 `driver_data` 成员得到设备私有信息结构体, 然后依次进行必要的硬件操作开始发送, 代码清单 14.6 为 `tty_driver` 的 `write()` 函数范例。

代码清单 14.6 `tty_driver` 结构体的 `write()` 成员函数范例

```
1 static int xxx_write(struct tty_struct *tty, const unsigned char *buf, int count)
2 {
3     /* 获得 tty 设备私有数据 */
4     struct xxx_tty *xxx = (struct xxx_tty*)tty->driver_data;
5     ...
6     /* 开始发送 */
7     while (1) {
8         local_irq_save(flags);
9         c = min_t(int, count, min(SERIAL_XMIT_SIZE - xxx->xmit_cnt - 1,
10             SERIAL_XMIT_SIZE - xxx->xmit_head));
11
12         if (c <= 0) {
13             local_irq_restore(flags);
14             break;
15         }
16         /* 拷贝到发送缓冲区 */
17         memcpy(xxx->xmit_buf + xxx->xmit_head, buf, c);
18         xxx->xmit_head = (xxx->xmit_head + c) & (SERIAL_XMIT_SIZE - 1);
```

```

19     xxx->xmit_cnt += c;
20     local_irq_restore(flags);
21
22     buf += c;
23     count -= c;
24     total += c;
25 }
26
27 if (xxx->xmit_cnt && !tty->stopped && !tty->hw_stopped)
28     start_xmit(xxx); /* 开始发送 */
29 return total; /* 返回发送的字节数 */
30 }

```

当 tty 子系统自己需要发送数据到 tty 设备时, 如果没有实现 `put_char()` 函数, `write()` 函数将被调用, 此时传入的 `count` 参数为 1, 通过对代码清单 14.7 的分析即可获知。

代码清单 14.7 `put_char()` 函数的 `write()` 替代

```

1 int tty_register_driver(struct tty_driver *driver)
2 {
3     ...
4     if (!driver->put_char) /* 没有定义 put_char() 函数 */
5         driver->put_char = tty_default_put_char;
6     ...
7 }
8 static void tty_default_put_char(struct tty_struct *tty, unsigned char ch)
9 {
10    tty->driver->write(tty, &ch, 1); /* 调用 tty_driver.write() 函数 */
11 }

```

读者朋友们可能注意到了, `tty_operations` 结构体中没有提供 `read()` 函数。因为发送是用户主动的, 而接收即用户调 `read()` 则是读一片缓冲区中已放好的数据。tty 核心在一个称为 `struct tty_flip_buffer` 的结构体中缓冲数据直到它被用户请求。因为 tty 核心提供了缓冲逻辑, 因此每个 tty 驱动并非一定要实现它自身的缓冲逻辑。

tty 驱动不必过于关心 `tty_flip_buffer` 结构体的细节, 如果其 `count` 字段大于或等于 `TTY_FLIPBUF_SIZE`, 这个 flip 缓冲区就需要被刷新到用户, 刷新通过对 `tty_flip_buffer_push()` 函数的调用来完成, 代码清单 14.8 给出了范例。

代码清单 14.8 `tty_flip_buffer.push()` 范例

```

1 for (i = 0; i < data_size; ++i) {
2     if (tty->flip.count >= TTY_FLIPBUF_SIZE)
3         tty_flip_buffer_push(tty); /* 数据填满向上层“推” */
4     tty_insert_flip_char(tty, data[i], TTY_NORMAL); /* 把数据插入缓冲区 */
5 }
6 tty_flip_buffer_push(tty);

```

从 tty 驱动接收到字符将被 `tty_insert_flip_char()` 函数插入 flip 缓冲区。该函数的第 1 个参数是数据应当保存入的 `tty_struct` 结构体, 第 2 个参数是要保存的字符, 第 3 个参数是应当为这个字符设置的标志, 如果字符是一个接收到的常规字符, 则设为 `TTY_NORMAL`, 如果是一个特殊类型的指示错误的字符, 依据具体的错误类型, 应当设为 `TTY_BREAK`、`TTY_PARITY` 或 `TTY_OVERRUN`。



14.5 TTY 线路设置

14.5.1 线路设置用户空间接口

用户可用如下两种方式改变 tty 设备的线路设置或者获取当前线路设置。

1. 调用用户空间的 termios 库函数

用户空间的应用程序需引用 `termios.h` 头文件, 该头文件包含了终端设备的 I/O 接口, 实际是由 POSIX 定义的标准方法。对终端设备操作模式的描述由 `termios` 结构体完成, 从代码清单 14.2 可以看出, 这个结构体包含 `c_iflag`、`c_oflag`、`c_cflag`、`c_lflag` 和 `c_cc[]` 几个成员。

`termios` 的 `c_cflag` 主要包含如下位域信息: `CSIZE` (字长)、`CSTOPB` (两个停止位)、`PARENB` (奇偶校验位使能)、`PARODD` (奇校验位, 当 `PARENB` 被使能时)、`CREAD` (字符接收使能, 如果没有置位, 仍然从端口接收字符, 但这些字符都要被丢弃)、`CRTSCTS` (如果被置位, 使能 CTS 状态改变报告)、`CLOCAL` (如果没有置位, 使能调制解调器状态改变报告)。

`termios` 的 `c_iflag` 主要包含如下位域信息: `INPCK` (使能帧和奇偶校验错误检查)、`BRKINT` (`break` 将清除终端输入/输出队列, 向该终端上前台的程序发出 `SIGINT` 信号)、`PARMRK` (奇偶校验和帧错误被标记, 在 `INPCK` 被设置且 `IGNPAR` 未被设置的情况下才有意义)、`IGNPAR` (忽略奇偶校验和帧错误)、`IGNBRK` (忽略 `break`)。

通过 `tcgetattr()`、`tcsetattr()` 函数即可完成对终端设备的操作模式的设置和获取, 这两个函数的原型如下:

```
int tcgetattr (int fd, struct termios *termios_p);
int tcsetattr (int fd, int optional_actions, struct termios *termios_p);
```

例如, `Raw` 模式的线路设置如下。

- 非正规模式。
- 关闭回显。
- 禁止 `CR` 到 `NL` 的映射 (`ICRNL`)、输入奇偶校验、输入第 8 位的截取 (`ISTRIP`) 以及输出流控制。
- 8 位字符 (`CS8`), 奇偶校验被禁止。
- 禁止所有的输出处理。
- 每次一个字节 (`c_cc [VMIN] = 1`、`c_cc [VTIME] = 0`)。

则对应的对 `termios` 结构体的设置就为:

```
termios_p->c_iflag &= ~(IGNBRK | BRKINT | PARMRK | ISTRIP
                      | INLCR | IGNCR | ICRNL | IXON);
termios_p->c_oflag &= ~OPOST;
termios_p->c_lflag &= ~(ECHO | ECHONL | ICANON | ISIG | IEXTEN);
termios_p->c_cflag &= ~(CSIZE | PARENB);
termios_p->c_cflag |= CS8;
```

通过如下一组函数可完成输入/输出波特率的获取和设置:

```
speed_t cfgetospeed (struct termios *termios_p);           // 获得输出波特率
speed_t cfgetispeed (struct termios *termios_p);           // 获得输入波特率
```

```
int cfsetospeed (struct termios *termios_p, speed_t speed); //设置输出波特率
int cfsetispeed (struct termios *termios_p, speed_t speed); //设置输入波特率
```

如下一组函数则完成线路控制：

```
int tcdrain (int fd); //等待所有输出都被发送
int tcflush (int fd, int queue_selector); //flush 输入/输出缓冲区
int tcflow (int fd, int action); //对输入和输出流进行控制
int tcsendbreak (int fd, int duration); //发送 break
```

tcflush 函数刷清（抛弃）输入缓冲区（终端驱动程序已接收到，但用户程序尚未读取）或输出缓冲区（用户程序已经写，但驱动尚未发送），**queue** 参数可取 **TCIFLUSH**（刷清输入队列）、**TCOFLUSH**（刷清输出队列）或 **TCIOFLUSH**（刷清输入、输出队列）。

tcflow()对输入输出进行流控制，**action** 参数可取 **TCOOFF**（输出被挂起）、**TCOON**（重新启动以前被挂起的输出）、**TCIOFF**（发送 1 个 STOP 字符，使终端设备暂停发送数据）、**TCION**（发送 1 个 START 字符，使终端恢复发送数据）。

tcsendbreak()函数在一个指定的时间区间内发送连续的二进位数 0。若 **duration** 参数为 0，则此种发送延续 0.25~0.5 秒。POSIX.1 说明若 **duration** 非 0，则发送时间依赖于实现。

2. 对 tty 设备节点进行 ioctl()调用

大部分 **termios** 库函数会被转化为对 **tty** 设备节点的 **ioctl()**调用，例如 **tcgetattr()**、**tcsetattr()**函数对应着 **TCGETS**、**TCSETS** IO 控制命令。

TIOCMGET（获得 MODEM 状态位）、**TIOCMSET**（设置 MODEM 状态位）、**TIOCMBIC**（清除指示 MODEM 位）、**TIOCMBIS**（设置指示 MODEM 位）这 4 个 I/O 控制命令用于获取和设置 MODEM 握手，如 RTS、CTS、DTR、DSR、RI、CD 等。

14.5.2 tty 驱动 set_termios 函数

大部分 **termios** 用户空间函数被库转换为对驱动节点的 **ioctl()**调用，而 **tty ioctl** 中的大部分命令会被 **tty** 核心转换为对 **tty** 驱动的 **set_termios()**函数的调用。**set_termios()**函数需要根据用户对 **termios** 的设置（**termios** 设置包括字长、奇偶校验位、停止位、波特率等）完成实际的硬件设置。

tty_operations 中的 **set_termios()**函数原型为：

```
void(*set_termios)(struct tty_struct *tty, struct termios *old);
```

新的设置被保存在 **tty_struct** 中，旧的设置被保存在 **old** 参数中，若新旧参数相同，则什么都不需要做，对于被改变的设置，需完成硬件上的设置，代码清单 14.9 给出了 **set_termios()**函数的例子。

代码清单 14.9 tty 驱动程序 set_termios()函数范例

```
1 static void xxx_set_termios(struct tty_struct *tty, struct termios *old_termios)
2 {
3     struct xxx_tty *info = (struct cyclades_port*)tty->driver_data;
4     /* 新设置等同于老设置，什么也不做 */
5     if (tty->termios->c_cflag == old_termios->c_cflag)
6         return ;
7     ...
8
9     /* 关闭 CRTSCTS 硬件流控制 */
10    if ((old_termios->c_cflag & CRTSCTS) && !(cflag & CRTSCTS)) {
11        ...
```



```
12 }
13
14 /* 打开 CRTSCTS 硬件流控制 */
15 if (!(old_termios->c_cflag & CRTSCTS) && (cflag & CRTSCTS)) {
16     ...
17 }
18
19 /* 设置字节大小 */
20 switch (tty->termios->c_cflag & CSIZE) {
21     case CS5:
22         ...
23     case CS6:
24         ...
25     case CS7:
26         ...
27     case CS8:
28         ...
29 }
30
31 /* 设置奇偶校验 */
32 if (tty->termios->c_cflag & PARENB)
33     if (tty->termios->c_cflag & PARODD) /* 奇校验 */
34         ...
35     else /* 偶校验 */
36         ...
37 else /* 无校验 */
38     ...
39 }
```

14.5.3 tty 驱动的 tiocmget 和 tiocmset 函数

对 TIOCMGET、TIOCMSET、TIOCMCBIC 和 TIOCMCBIS IO 控制命令的调用将被 tty 核心转换为对 tty 驱动 tiocmget()函数和 tiocmset()函数的调用, TIOCMGET 对应 tiocmget()函数, TIOCMSET、TIOCMCBIC 和 TIOCMCBIS 对应 tiocmset()函数, 分别用于读取 Modem 控制的设置和进行 Modem 的设置。代码清单 14.10 所示为 tiocmget()函数的范例, 代码清单 14.11 所示为 tiocmset()函数的范例。

代码清单 14.10 tty 驱动程序 tiocmget()函数范例

```
1 static int xxx_tiocmget(struct tty_struct *tty, struct file *file)
2 {
3     struct xxx_tty *info = tty->driver_data;
4     unsigned int result = 0;
5     unsigned int msr = info->msr;
6     unsigned int mcr = info->mcr;
7     result = ((mcr & MCR_DTR) ? TIOCM_DTR : 0) | /* DTR 被设置 */
8     ((mcr & MCR_RTS) ? TIOCM_RTS : 0) | /* RTS 被设置 */
9     ((mcr & MCR_LOOP) ? TIOCM_LOOP : 0) | /* LOOP 被设置 */
10    ((msr & MSR_CTS) ? TIOCM_CTS : 0) | /* CTS 被设置 */
11    ((msr & MSR_CD) ? TIOCM_CAR : 0) | /* CD 被设置 */
12    ((msr & MSR_RI) ? TIOCM_RI : 0) | /* 振铃指示被设置 */
13    ((msr & MSR_DSR) ? TIOCM_DSR : 0); /* DSR 被设置 */
14     return result;
15 }
```


代码清单 14.11 tty 驱动程序 tiocmset()函数范例

```

1 static int xxx_tiocmset(struct tty_struct *tty, struct file *file, unsigned
2   int set, unsigned int clear)
3 {
4   struct xxx_tty *info = tty->driver_data;
5   unsigned int mcr = info->mcr;
6
7   if (set & TIOCM_RTS) /* 设置 RTS */
8     mcr |= MCR_RTS;
9   if (set & TIOCM_DTR) /* 设置 DTR */
10    mcr |= MCR_RTS;
11
12   if (clear & TIOCM_RTS) /* 清除 RTS */
13     mcr &= ~MCR_RTS;
14   if (clear & TIOCM_DTR) /* 清除 DTR */
15     mcr &= ~MCR_RTS;
16
17   /* 设置设备新的 MCR 值 */
18   tiny->mcr = mcr;
19   return 0;
20 }

```

tiocmget()函数会访问 MODEM 状态寄存器 (MSR)，而 tiocmset()函数会访问 MODEM 控制寄存器 (MCR)。

14.5.4 tty 驱动 ioctl 函数

当用户在 tty 设备节点上进行 ioctl()调用时，tty_operations 中的 ioctl()函数会被 tty 核心调用。如果 tty 驱动不知道如何处理传递给它的 IOCTL 值，它返回-ENOIOCTLCMD，之后 tty 核心会执行一个通用的操作。

驱动中常见的需处理的 I/O 控制命令包括 TIOCSERGETLSR (获得这个 tty 设备的线路状态寄存器 LSR 的值)、TIOCGSERIAL (获得串口线信息)、TIOCMIWAIT (等待 MSR 改变)、TIOCGICOUNT (获得中断计数) 等。代码清单 14.12 给出了 tty 驱动程序 ioctl()函数的范例。

代码清单 14.12 tty 驱动程序 ioctl()函数范例

```

1 static int xxx_ioctl(struct tty_struct *tty, struct file *filp, unsigned int
2   cmd, unsigned long arg)
3 {
4   struct xxx_tty *info = tty->driver_data;
5   ...
6   /* 处理各种命令 */
7   switch (cmd){
8   case TIOCGSERIAL:
9     ...
10  case TIOCSSERIAL:
11    ...
12  case TIOCSERCONFIG:
13    ...
14  case TIOCMIWAIT:
15    ...
16  case TIOCGICOUNT:

```



```
17    ...
18    case TIOCSERGETLSR:
19    ...
20    }
21    ...
22 }
```

14.6 UART 设备驱动

尽管一个特定的 UART 设备驱动完全可以遵循第 14.2~14.5 节的方法来设计，即定义 `tty_driver` 并实现 `tty_operations` 其中的成员函数，但是 Linux 已经在文件 `serial_core.c` 中实现了 UART 设备的通用 `tty` 驱动层（姑且称其为串口核心层），这样，UART 驱动的主要任务演变成实现 `serial-core.c` 中定义的一组 `uart_xxx` 接口而非 `tty_xxx` 接口，如图 14.4 所示。

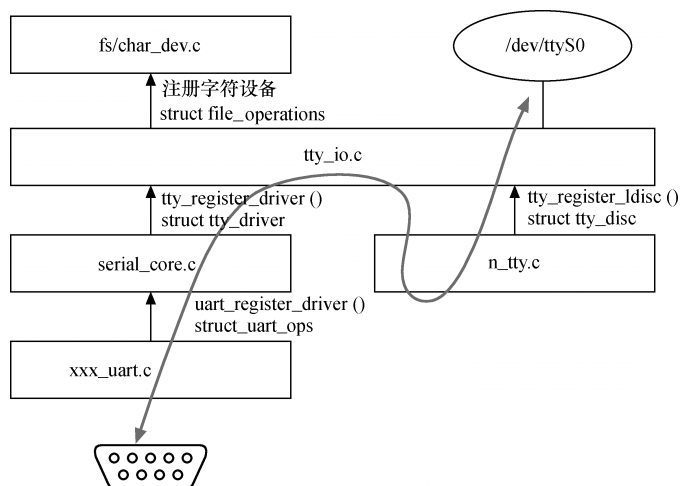


图 14.4 串口核心层

`serial_core.c` 串口核心层完全可以被当作 14.2~14.5 节 `tty` 设备驱动的实例，它实现了 UART 设备的 `tty` 驱动。回过头来再看 12.2 节“设备驱动的分层思想”，是否更加豁然开朗？

串口核心层为串口设备驱动提供了如下 3 个结构体。

1. `uart_driver`

`uart_driver` 包含串口设备的驱动名、设备名、设备号等信息，它封装了 `tty_driver`，使得底层的 UART 驱动无需关心 `tty_driver`，其定义如代码清单 14.13 所示。

代码清单 14.13 `uart_driver` 结构体

```
1 struct uart_driver {
2     struct module *owner;
3     const char *driver_name; /* 驱动名 */
4     const char *dev_name;   /* 设备名 */
5     int major; /* 主设备号 */
6     int minor; /* 次设备号 */
```

```

7   int nr;
8   struct console *cons;
9
10  /* 私有的，底层驱动不应该访问这些成员，应该被初始化为 NULL */
11  struct uart_state *state;
12  struct tty_driver *tty_driver;
13 };

```

一个 tty 驱动必须注册/注销 tty_driver，而一个 UART 驱动则演变为注册/注销 uart_driver，使用如下接口：

```

int uart_register_driver(struct uart_driver *drv);
void uart_unregister_driver(struct uart_driver *drv);

```

实际上，uart_register_driver()和 uart_unregister_driver()中分别包含了 tty_register_driver()和 tty_unregister_driver()的操作，如代码清单 14.14 所示。

代码清单 14.14 uart_register_driver()和 uart_unregister_driver()函数

```

1  int uart_register_driver(struct uart_driver *drv)
2  {
3      struct tty_driver *normal = NULL;
4      int i, retval;
5
6      BUG_ON(drv->state);
7
8      drv->state = kzalloc(sizeof(struct uart_state) * drv->nr, GFP_KERNEL);
9      retval = -ENOMEM;
10     if (!drv->state)
11         goto out;
12
13     normal = alloc_tty_driver(drv->nr);
14     if (!normal)
15         goto out;
16
17     drv->tty_driver = normal;
18     /* 初始化 tty_driver */
19     normal->owner      = drv->owner;
20     normal->driver_name = drv->driver_name;
21     normal->name        = drv->dev_name;
22     normal->major       = drv->major;
23     normal->minor_start = drv->minor;
24     normal->type        = TTY_DRIVER_TYPE_SERIAL;
25     normal->subtype     = SERIAL_TYPE_NORMAL;
26     normal->init_termios = tty_std_termios;
27     normal->init_termios.c_cflag = B9600 | CS8 | CREAD | HUPCL | CLOCAL;
28     normal->init_termios.c_ispeed = normal->init_termios.c_ospeed = 9600;
29     normal->flags = TTY_DRIVER_REAL_RAW | TTY_DRIVER_DYNAMIC_DEV;
30     normal->driver_state = drv;
31     tty_set_operations(normal, &uart_ops);
32
33     /* 初始化 UART 状态 */
34     for (i = 0; i < drv->nr; i++) {
35         struct uart_state *state = drv->state + i;
36
37         state->close_delay = 500; /* .5 seconds */
38         state->closing_wait = 30000; /* 30 seconds */

```



```
39
40     mutex_init(&state->mutex);
41 }
42
43     retval = tty_register_driver(normal);
44 out:
45     if (retval < 0) {
46         put_tty_driver(normal);
47         kfree(drv->state);
48     }
49     return retval;
50 }
51
52 void uart_unregister_driver(struct uart_driver *drv)
53 {
54     struct tty_driver *p = drv->tty_driver;
55     tty_unregister_driver(p);
56     put_tty_driver(p);
57     kfree(drv->state);
58     drv->tty_driver = NULL;
59 }
```

2. uart_port

uart_port 用于描述一个 UART 端口（直接对应于一个串口）的 I/O 端口或 I/O 内存地址、FIFO 大小、端口类型等信息，其定义如代码清单 14.15。

代码清单 14.15 uart_port 结构体

```
1 struct uart_port {
2     spinlock_t lock; /* 端口锁 */
3     unsigned int iobase; /* I/O 端口基地址 */
4     unsigned char __iomem *membase; /* I/O 内存基地址 */
5     unsigned int irq; /* 终端号 */
6     unsigned int uartclk; /* UART 时钟 */
7     unsigned char fifosize; /* 传输 fifo 大小 */
8     unsigned char x_char; /* xon/xoff 字符 */
9     unsigned char regshift; /* 寄存器位移 */
10    unsigned char iotype; /* I/O 存取类型 */
11    unsigned char      unused1;
12
13    unsigned int read_status_mask; /* 驱动相关的 */
14    unsigned int ignore_status_mask; /* 驱动相关的 */
15    struct uart_info *info; /* 指向 parent 信息 */
16    struct uart_icount icount; /* 计数 */
17
18    struct console *cons; /* console 结构体 */
19    #ifdef CONFIG_SERIAL_CORE_CONSOLE
20        unsigned long sysrq; /* sysrq 超时 */
21    #endif
22
23    upf_t flags;
24    unsigned int mctrl; /* 目前 modem 控制设置 */
25    unsigned int timeout; /* 基于字符的超时 */
26    unsigned int type; /* 端口类型 */
27    const struct uart_ops *ops; /* UART 操作集 */
28 }
```

```

28 unsigned int custom_divisor;
29 unsigned int line; /* 端口索引 */
30 unsigned long mapbase; /* ioremap 后基地址 */
31 struct device *dev; /* parent 设备 */
32 unsigned char hub6;
33 unsigned char suspended;
34 unsigned char unused[2];
35 void *private_data;
36 };

```

串口核心层提供如下函数来添加一个端口：

```
int uart_add_one_port(struct uart_driver *drv, struct uart_port *port);
```

对上述函数的调用应该发生在 `uart_register_driver()` 之后，`uart_add_one_port()` 的一个最重要作用是封装了 `tty_register_device()`。

`uart_add_one_port()` 的“反函数”是 `uart_remove_one_port()`，其中会调用 `tty_unregister_device()`，原型为：

```
int uart_remove_one_port(struct uart_driver *drv, struct uart_port *port);
```

3. uart_ops

`uart_ops` 定义了针对 UART 的一系列操作，包括发送、接收及线路设置等，如果说 `tty_driver` 中的 `tty_operations` 对于串口还较为抽象，那么 `uart_ops` 则直接面向了串口的 UART，其定义如代码清单 14.16。Linux 驱动的这种层次非常类似于面向对象编程中基类、派生类的关系，派生类针对特定的事物会更加具体，而基类则站在更高的抽象层次上。

代码清单 14.16 `uart_ops` 结构体

```

1 struct uart_ops {
2     unsigned int      (*tx_empty)(struct uart_port *);
3     void              (*set_mctrl)(struct uart_port *, unsigned int mctrl);
4     unsigned int      (*get_mctrl)(struct uart_port *);
5     void              (*stop_tx)(struct uart_port *);
6     void              (*start_tx)(struct uart_port *);
7     void              (*send_xchar)(struct uart_port *, char ch);
8     void              (*stop_rx)(struct uart_port *);
9     void              (*enable_ms)(struct uart_port *);
10    void              (*break_ctl)(struct uart_port *, int ctl);
11    int               (*startup)(struct uart_port *);
12    void              (*shutdown)(struct uart_port *);
13    void              (*flush_buffer)(struct uart_port *);
14    void              (*set_termios)(struct uart_port *, struct ktermios *new,
15                                   struct ktermios *old);
16    void              (*set_ldisc)(struct uart_port *);
17    void              (*pm)(struct uart_port *, unsigned int state,
18                           unsigned int oldstate);
19    int               (*set_wake)(struct uart_port *, unsigned int state);
20
21    const char *(*type)(struct uart_port *); /* 一个描述端口类型的字符串 */
22    /* 释放该端口使用的 I/O 和 memory 资源 */
23    void              (*release_port)(struct uart_port *);
24
25    /* 申请该端口使用的 I/O 和 memory 资源 */
26    int               (*request_port)(struct uart_port *);
27    void              (*config_port)(struct uart_port *, int);

```



```
28     int      (*verify_port)(struct uart_port *, struct serial_struct *);
29     int      (*ioctl)(struct uart_port *, unsigned int, unsigned long);
30 #ifdef CONFIG_CONSOLE_POLL
31     void      (*poll_put_char)(struct uart_port *, unsigned char);
32     int      (*poll_get_char)(struct uart_port *);
33 #endif
34 };
```

serial_core.c 中定义了 tty_operations 的实例, 包含 uart_open()、uart_close()、uart_write()、uart_send_xchar()等成员函数 (如代码清单 14.17), 这些函数会借助 uart_ops 结构体中的成员函数来完成具体的操作, 代码清单 14.18 所示为 tty_operations 的 uart_send_xchar()成员函数利用 uart_ops 中 start_tx()、send_xchar()成员函数的例子。

代码清单 14.17 串口核心层的 tty_operations 实例

```
1 static const struct tty_operations uart_ops = {
2     .open          = uart_open,
3     .close         = uart_close,
4     .write         = uart_write,
5     .put_char      = uart_put_char,
6     .flush_chars   = uart_flush_chars,
7     .write_room    = uart_write_room,
8     .chars_in_buffer= uart_chars_in_buffer,
9     .flush_buffer  = uart_flush_buffer,
10    .ioctl          = uart_ioctl,
11    .throttle       = uart_throttle,
12    .unthrottle     = uart_unthrottle,
13    .send_xchar     = uart_send_xchar,
14    .set_termios    = uart_set_termios,
15    .set_ldisc      = uart_set_ldisc,
16    .stop           = uart_stop,
17    .start          = uart_start,
18    .hangup         = uart_hangup,
19    .break_ctl      = uart_break_ctl,
20    .wait_until_sent= uart_wait_until_sent,
21 #ifdef CONFIG_PROC_FS
22    .read_proc      = uart_read_proc,
23 #endif
24    .tiocmget       = uart_tiocmget,
25    .tiocmset       = uart_tiocmset,
26 #ifdef CONFIG_CONSOLE_POLL
27    .poll_init      = uart_poll_init,
28    .poll_get_char  = uart_poll_get_char,
29    .poll_put_char  = uart_poll_put_char,
30 #endif
31 };
```

代码清单 14.18 串口核心层的 tty_operations 与 uart_ops 关系

```
1 static void uart_send_xchar(struct tty_struct *tty, char ch)
2 {
3     struct uart_state *state = tty->driver_data;
4     struct uart_port *port = state->port;
5     unsigned long flags;
6     /* 如果 uart_ops 中实现了 send_xchar 成员函数 */
7     if (port->ops->send_xchar)
```

```

8     port->ops->send_xchar(port, ch);
9     else { /* uart_ops 中未实现 send_xchar 成员函数 */
10        port->x_char = ch;
11        if (ch) {
12            spin_lock_irqsave(&port->lock, flags);
13            port->ops->start_tx(port); /* 发送 xchar */
14            spin_unlock_irqrestore(&port->lock, flags);
15        }
16    }
17 }

```

在使用串口核心层这个通用串口 tty 驱动层的接口后，一个串口驱动要完成的主要工作如下。

(1) 定义 `uart_driver`、`uart_ops`、`uart_port` 等结构体的实例并在适当的地方根据具体硬件和驱动的情况初始化它们，当然具体设备 `xxx` 的驱动可以将这些结构套在新定义的 `xxx_uart_driver`、`xxx_uart_ops`、`xxx_uart_port` 之内。

(2) 在模块初始化时调用 `uart_register_driver()` 和 `uart_add_one_port()` 以注册 UART 驱动并添加端口，在模块卸载时调用 `uart_unregister_driver()` 和 `uart_remove_one_port()` 以注销 UART 驱动并移除端口。

(3) 根据具体硬件的 `datasheet` 实现 `uart_ops` 中的成员函数，这些函数的实现成为 UART 驱动的主体工作。

14.7 printk 和 early_printk console 驱动

在 Linux 内核中，`printk()` 函数是最常用的调试手段。`printk()` 的打印消息会放入一个环形缓冲区，而 `/proc/kmsg` 文件用于描述这个缓冲区。通过 `dmesg` 命令或 `klogd` 可以读取该缓冲区。如果用户空间的 `klogd` 守护进程在运行，它将获取内核消息并分发给 `syslogd`，`syslogd` 接着检查 `/etc/syslog.conf` 来找出如何处理它们。

内核 `printk` 信息支持 8 个级别，从高到低分别是：`KERN_EMERG`、`KERNEL_ALERT`、`KERN_CRIT`、`KERN_ERR`、`KERN_WARNING`、`KERN_NOTICE`、`KERN_INFO`、`KERN_DEBUG`。当调用 `printk()` 函数时指定的优先级小于指定的控制台优先级 `console_loglevel` 时，调试消息就显示在控制台终端。缺省的 `console_loglevel` 值是 `DEFAULT_CONSOLE_LOGLEVEL`，用户可以使用系统调用 `sys_syslog` 或 `klogd-c` 来修改 `console_loglevel` 值，也可以直接 `echo` 值到 `/proc/sys/kernel/printk`。`/proc/sys/kernel/printk` 文档包含 4 个整数值，前两个表示系统当前的优先级和缺省优先级。

在 Linux 中，用于 `printk` 输出的是内核 `console`，专门用 `console` 结构体来描述，如代码清单 14.19 所示。

代码清单 14.19 用于 `printk` 的 `console` 结构体

```

1 struct console {
2     char    name[16];
3     void    (*write)(struct console *, const char *, unsigned);
4     int     (*read)(struct console *, char *, unsigned);
5     struct tty_driver *(*device)(struct console *, int *);

```



```
6      void      (*unblank) (void);
7      int       (*setup) (struct console *, char *);
8      int       (*early_setup) (void);
9      short     flags;
10     short     index;
11     int       cflag;
12     void      *data;
13     struct    console *next;
14 };
```

其中, 较关键的是 `write()` 和 `setup()` 成员函数, 前者用于将打印消息写入 `console`, 后者用于设置 `console` 的特性, 如波特率、停止位等。

`printk()` 函数经过重重调用, 经过 `__call_console_drivers()` 函数, 最终调用 `console` 的 `write()` 成员函数将控制台消息打印出去, 如代码清单 14.20 所示。

代码清单 14.20 `printk()` 最终调用到 `console` 的 `write()` 成员函数

```
1 static void __call_console_drivers(unsigned start, unsigned end)
2 {
3     struct console *con;
4
5     for (con = console_drivers; con; con = con->next) {
6         if ((con->flags & CON_ENABLED) && con->write &&
7             (cpu_online(smp_processor_id()) ||
8              (con->flags & CON_ANYTIME)))
9             con->write(con, &LOG_BUF(start), end - start);
10    }
11 }
```

内核提供如下 API 用于注册和注销 `console`:

```
void register_console(struct console *);
int unregister_console(struct console *);
```

在内核 `init/main.c` 文件中的 `start_kernel()` 函数中, 会调用 `console_init()` 函数, 该函数会调用位于内核存放 `console` 初始化函数的代码段, 调用其中的每一个初始化 `console` 的函数, 如代码清单 14.21。

代码清单 14.21 `console_init()` 函数

```
1 void __init console_init(void)
2 {
3     initcall_t *call;
4
5     tty_ldisc_begin();
6
7     call = __con_initcall_start;
8     while (call < __con_initcall_end) {
9         (*call)();
10        call++;
11    }
12 }
```

实际上, 对于任何一个初始化 `console` 的函数而言, 只需要通过 `console_initcall()` 进行包装, 即可把它放入 `con_initcall.init` 节 (开始地址为 `__con_initcall_start`), 典型地, 如最常用的 8250 对应的 `console` 结构体以及初始化代码如清单 14.22。

代码清单 14.22 8250 的 `console` 及 `console_initcall`

```
1 static struct console serial8250_console = {
2     .name      = "ttyS",
```



```

3     .write      = serial8250_console_write,
4     .device     = uart_console_device,
5     .setup      = serial8250_console_setup,
6     .early_setup = serial8250_console_early_setup,
7     .flags      = CON_PRINTBUFFER,
8     .index      = -1,
9     .data       = &serial8250_reg,
10  };
11
12  static int __init serial8250_console_init(void)
13  {
14      if (nr_uarts > UART_NR)
15          nr_uarts = UART_NR;
16
17      serial8250_isa_init_ports();
18      register_console(&serial8250_console);
19      return 0;
20  }
21  console_initcall(serial8250_console_init);

```

实际上，`console_initcall()`是一个宏，其定义于 `include/linux/init.h` 文件，它可以展开成：

```

#define console_initcall(fn) \
    static initcall_t __initcall_##fn \
    __used __section(.con_initcall.init) = fn

```

留意其中的 `__section(.con_initcall.init)`，实际上是一个链接阶段的指示，表明将指定的函数放入 `.con_initcall.init` 节。

`console_init()`是由 `init/main.c` 文件中的 `start_kernel()`函数调用的，而在 `console_init()`被调用前，还执行了一系列的操作。为了在 `console_init()`被调用前就能使用 `printk()`，可以使用内核的“early printk”支持，该选项位于内核配置菜单“Linux Kernel Configuration”下的“Kernel hacking”菜单之下。

对于 early printk 的 console 的注册往往通过解析内核的 `early_param` 完成，如对于 8250 而言，定义了“earlycon”这样一个内核参数，当解析此内核参数时，相应地被 `early_param()`绑定的函数 `setup_early_serial8250_console()`被调用，此函数将注册一个用于 early printk 的 console。

代码清单 14.23 8250 的 early printk console

```

1  static struct console early_serial8250_console __initdata = {
2      .name      = "uart",
3      .write     = early_serial8250_write,
4      .flags     = CON_PRINTBUFFER | CON_BOOT,
5      .index     = -1,
6  };
7
8  int __init setup_early_serial8250_console(char *cmdline)
9  {
10     char *options;
11     int err;
12
13     options = strstr(cmdline, "uart8250,");
14     if (!options) {
15         options = strstr(cmdline, "uart,");
16         if (!options)
17             return 0;
18     }
19
20     options = strchr(cmdline, ',') + 1;
21     err = early_serial8250_setup(options);

```



```
22     if (err < 0)
23         return err;
24
25     register_console(&early_serial8250_console);
26
27     return 0;
28 }
29
30 early_param("earlycon", setup_early_serial8250_console);
```

例如,在 Linux 启动的 command line 中设置如下参数,将使能 8250 作为 early printk 的 console。

```
earlycon=uart8250,mmio,0xff5e0000,115200n8
earlycon=uart8250,io,0x3f8,9600n8
```

留意一下,代码清单 14.22 第 7 行的 flags 和代码清单 14.23 第 4 行的 flags 的区别,会发现后者多出了一个 CON_BOOT 属性。实例上,所有的具有 CON_BOOT 属性的 console 都会在内核初始化至 late initcall 阶段的时候被注销,注销它们的函数是 disable_boot_consoles(),其定义如代码清单 14.24。

代码清单 14.24 disable_boot_consoles()函数

```
1 static int __init disable_boot_consoles(void)
2 {
3     if (console_drivers != NULL) {
4         if (console_drivers->flags & CON_BOOT) {
5             printk(KERN_INFO "turn off boot console %s%d\n",
6                     console_drivers->name, console_drivers->index);
7             return unregister_console(console_drivers);
8         }
9     }
10    return 0;
11 }
12 late_initcall(disable_boot_consoles);
```

这里再补充一个知识,内核的 initcall 分成了 8 级,对应的节分别为 .initcall0.init、.initcall1.init、.initcall2.init、.initcall3.init、.initcall4.init、.initcall5.init、.initcall6.init、.initcall7.init,分别通过 pure_initcall(fn)、core_initcall(fn)、postcore_initcall(fn)、arch_initcall(fn)、subsys_initcall(fn)、fs_initcall(fn)、device_initcall(fn)、late_initcall(fn) 可将指定的函数放入对应的节。对于 pure_initcall() 而言,它指定的 initcall 不依赖于任何其他部分,因此,其指定函数只能 built-in,不能在模块中。对于 1~7 级而言,还存在对应的 sync 版本,分别通过 core_initcall_sync(fn)、postcore_initcall_sync(fn)、arch_initcall_sync(fn)、subsys_initcall_sync(fn)、fs_initcall_sync(fn)、device_initcall_sync(fn)、late_initcall_sync(fn) 修饰。

通过代码清单 14.24 的第 12 行可以看出,disable_boot_consoles() 被 late_initcall() 修饰,因此被放入了 .initcall7.init 这个节。

14.8 实例：S3C6410 串口与 console 驱动

14.8.1 S3C6410 串口硬件描述

S3C6410 具有 4 路高速串口, LDD6410 为第 1 路串口设计了 1 个 DB9 接口,其他都通过 DIP 插座引出,其原理如图 14.5 所示,图中的 SP3232EEA 是 1 个电平转换芯片。

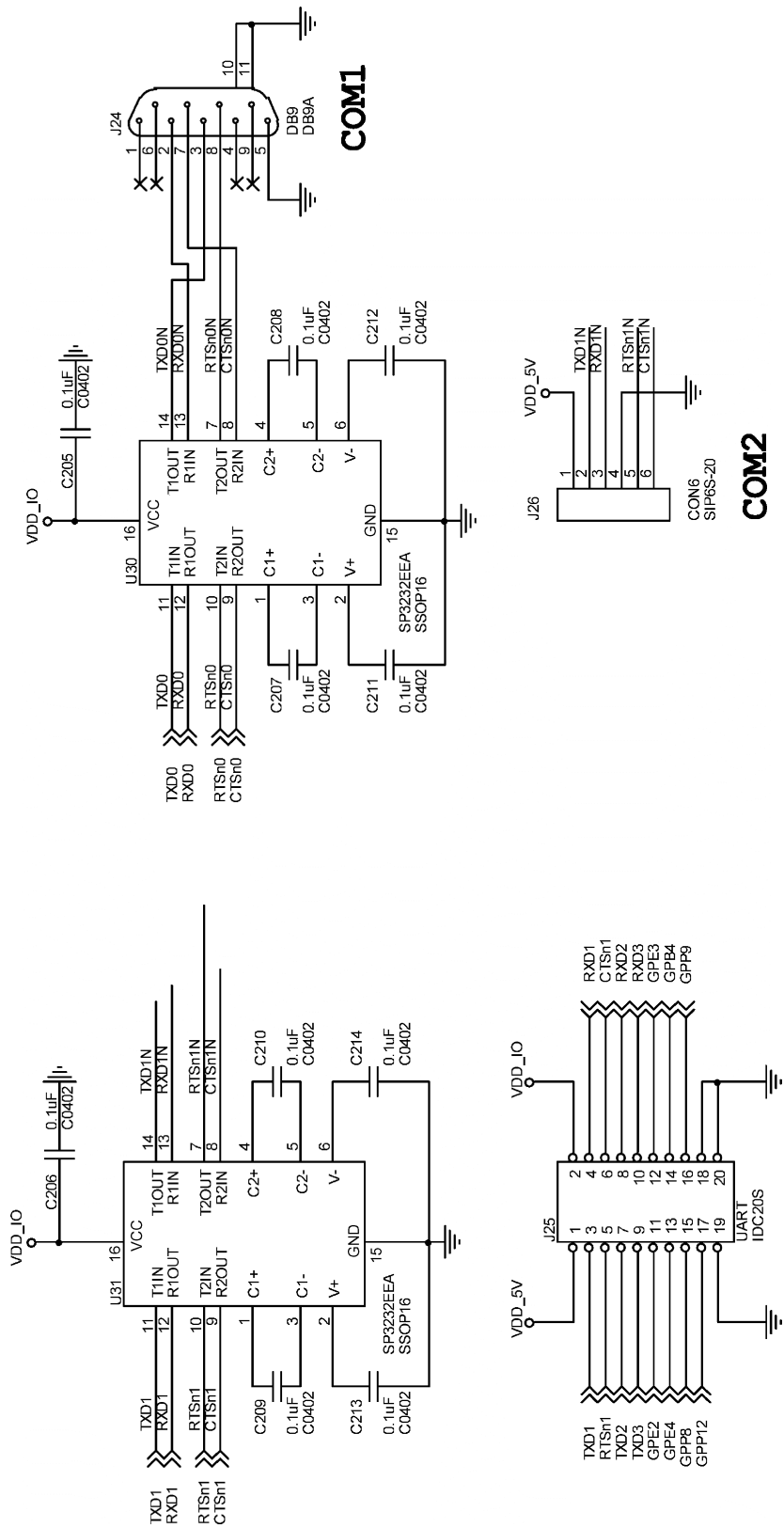


图 14.5 LDD6410 串口连接



LDD6410 开发板所用内核版本 2.6.28.6 的串口驱动位于 `drivers/serial/samsung.c`、`drivers/serial/s3c6400.c` 文件, 其中 `samsung.c` 是底层核心, 而 `s3c6400.c` (为 S3C6400、S3C6410、CPU_S5P644 服务) 调用了它。

14.8.2 S3C6410 串口 UART 驱动

在 `samsung.c` 的模块加载和卸载函数中, 通过 `uart_register_driver()`、`uart_unregister_driver()` 注册和注销了 `s3c24xx_uart_drv`, 如代码清单 14.25。

代码清单 14.25 S3C6410 串口驱动核心模块加载与卸载函数

```
1 static struct uart_driver s3c24xx_uart_drv = {
2     .owner          = THIS_MODULE,
3     .dev_name       = "s3c2410_serial",
4     .nr             = 3,
5     .cons           = S3C24XX_SERIAL_CONSOLE,
6     .driver_name     = S3C24XX_SERIAL_NAME,
7     .major          = S3C24XX_SERIAL_MAJOR,
8     .minor          = S3C24XX_SERIAL_MINOR,
9 };
10
11 static int __init s3c24xx_serial_modinit(void)
12 {
13     int ret;
14
15     ret = uart_register_driver(&s3c24xx_uart_drv);
16     if (ret < 0) {
17         printk(KERN_ERR "failed to register UART driver\n");
18         return -1;
19     }
20
21     return 0;
22 }
23
24 static void __exit s3c24xx_serial_modexit(void)
25 {
26     uart_unregister_driver(&s3c24xx_uart_drv);
27 }
28
29 module_init(s3c24xx_serial_modinit);
30 module_exit(s3c24xx_serial_modexit);
```

`s3c6400.c` 是一个 platform 驱动, 在其 `probe()` 成员函数 `s3c6400_serial_probe()` 中, 会调用 `samsung.c` 中的 `s3c24xx_serial_probe()`, 而该函数会添加 `uart_port`:

```
int s3c24xx_serial_probe(struct platform_device *dev,
                        struct s3c24xx_uart_info *info)
{
    ...
    uart_add_one_port(&s3c24xx_uart_drv, &ourport->port);
    ...
}
```

相反的, 在 `s3c6400.c` 这一 platform 驱动的 `remove()` 成员函数 `s3c24xx_serial_remove()` 中, 会调用 `uart_remove_one_port()` 去除 `uart_port`。

注意被添加的 `uart_port` 的 `uart_ops` 成员定义在 `samsung.c` 文件中，如代码清单 12.26。

代码清单 12.26 S3C6410 串口驱动的 `uart_ops`

```
1 static struct uart_ops s3c24xx_serial_ops = {
2     .pm      = s3c24xx_serial_pm,
3     .tx_empty    = s3c24xx_serial_tx_empty,
4     .get_mctrl   = s3c24xx_serial_get_mctrl,
5     .set_mctrl   = s3c24xx_serial_set_mctrl,
6     .stop_tx     = s3c24xx_serial_stop_tx,
7     .start_tx    = s3c24xx_serial_start_tx,
8     .stop_rx     = s3c24xx_serial_stop_rx,
9     .enable_ms   = s3c24xx_serial_enable_ms,
10    .break_ctl    = s3c24xx_serial_break_ctl,
11    .startup      = s3c24xx_serial_startup,
12    .shutdown     = s3c24xx_serial_shutdown,
13    .set_termios  = s3c24xx_serial_set_termios,
14    .type         = s3c24xx_serial_type,
15    .release_port = s3c24xx_serial_release_port,
16    .request_port = s3c24xx_serial_request_port,
17    .config_port  = s3c24xx_serial_config_port,
18    .verify_port  = s3c24xx_serial_verify_port,
19    #if defined(CONFIG_S5P_UART_DMA_EN)
20    .flush_buffer = s3c24xx_flush_buffer,
21    #endif
22 };
```

14.8.3 S3C6410 串口 console 驱动

在使能内核配置选项 `CONFIG_SERIAL_SAMSUNG_CONSOLE` 的情况下，S3C6410 串口驱动的 `console` 部分会被包含，它位于 `drivers/serial/samsung.c`，如代码清单 16.27 所示。

代码清单 16.27 S3C6410 串口 console 驱动

```
1 static struct console s3c24xx_serial_console = {
2     .name      = S3C24XX_SERIAL_NAME,
3     .device    = uart_console_device,
4     .flags     = CON_PRINTBUFFER,
5     .index     = -1,
6     .write     = s3c24xx_serial_console_write,
7     .setup     = s3c24xx_serial_console_setup
8 };
9
10 int s3c24xx_serial_initconsole(struct platform_driver *drv,
11                               struct s3c24xx_uart_info *info)
12 {
13     {
14         struct platform_device *dev = s3c24xx_uart_devs[0];
15
16         ...
17         if (strcmp(dev->name, drv->driver.name) != 0)
18             return 0;
19
20         s3c24xx_serial_console.data = &s3c24xx_uart_drv;
21         s3c24xx_serial_init_ports(info);
22     }
```



```
30
31     register_console(&s3c24xx_serial_console);
32     return 0;
33 }
```

而在 `drivers/serial/samsung.h` 文件中, 通过如下方法将 `s3c24xx_serial_initconsole` 放入了 `.con_initcall.init` 代码段。这样 `console_init()` 即会调用该函数。

14.9 总结

TTY 设备驱动的主体工作围绕 `tty_driver` 这个结构体的成员函数展开, 主要应实现其中的数据发送和接收流程以及 tty 设备线路设置接口函数。

针对串口, 内核实现了串口核心层, 这个层实现了串口设备通用的 `tty_driver`。因此, 串口设备驱动的主体工作从 `tty_driver` 转移到了 `uart_driver`。

LINUX

第15章

Linux 的 I²C 核心、总线与设备驱动

I²C 总线仅仅使用 SCL、SDA 这两根信号线就实现了设备之间的数据交互，极大地简化了对硬件资源和 PCB 板布线空间的占用。因此，I²C 总线被非常广泛地应用在 EEPROM、实时钟、小型 LCD 等设备与 CPU 的接口中。

Linux 系统定义了 I²C 驱动体系结构，在 Linux 系统中，I²C 驱动由 3 部分组成，即 I²C 核心、I²C 总线驱动和 I²C 设备驱动。这 3 部分相互协作，形成了非常通用、可适应性很强的 I²C 框架。

6.1 节对 Linux I²C 体系结构进行分析，讲解 3 个组成部分各自的功能及相互联系。

6.2 节对 Linux I²C 核心进行分析，讲解 i2c-core.c 文件的功能和主要函数的实现。

6.3 节、6.4 节分别详细介绍 I²C 总线驱动和 I²C 设备驱动的编写方法，给出可供参考的设计模板。

6.5 节、6.6 节以 6.3 节和 6.4 节给出的设计模板为基础，讲解 S3C6410 ARM 处理器 I²C 总线驱动以及挂接在 I²C 总线上的 AT24XX 系列 EEPROM 驱动。





15.1 Linux 的 I²C 体系结构

Linux 的 I²C 体系结构分为 3 个组成部分。

(1) I²C 核心。

I²C 核心提供了 I²C 总线驱动和设备驱动的注册、注销方法, I²C 通信方法 (即 “algorithm”) 上层的、与具体适配器无关的代码以及探测设备、检测设备地址的上层代码等。

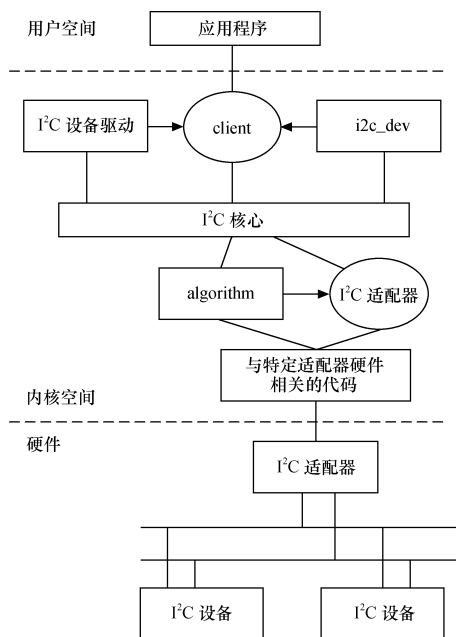


图 15.1 Linux I²C 体系结构

(2) I²C 总线驱动。

I²C 总线驱动是对 I²C 硬件体系结构中适配器端的实现, 适配器可由 CPU 控制, 甚至可以直接集成在 CPU 内部。

I²C 总线驱动主要包含了 I²C 适配器数据结构 `i2c_adapter`、I²C 适配器的 `algorithm` 数据结构 `i2c_algorithm` 和控制 I²C 适配器产生通信信号的函数。

经由 I²C 总线驱动的代码, 我们可以控制 I²C 适配器以主控方式产生开始位、停止位、读写周期, 以及以从设备方式被读写、产生 ACK 等。

(3) I²C 设备驱动。

I²C 设备驱动 (也称为客户驱动) 是对 I²C 硬件体系结构中设备端的实现, 设备一般挂接在受 CPU 控制的 I²C 适配器上, 通过 I²C 适配器与 CPU 交换数据。

I²C 设备驱动主要包含了数据结构 `i2c_driver` 和 `i2c_client`, 我们需要根据具体设备实现其中的成员函数。

在 Linux 2.6 内核中, 所有的 I²C 设备都在 `sysfs` 文件系统中显示, 存于 `/sys/bus/i2c/` 目录, 以

适配器地址和芯片地址的形式列出，例如：

```
$ tree /sys/bus/i2c/
/sys/bus/i2c/
|-- devices
| |-- 0-0048 -> ../../../../devices/legacy/i2c-0/0-0048
| |-- 0-0049 -> ../../../../devices/legacy/i2c-0/0-0049
| |-- 0-004a -> ../../../../devices/legacy/i2c-0/0-004a
| |-- 0-004b -> ../../../../devices/legacy/i2c-0/0-004b
| |-- 0-004c -> ../../../../devices/legacy/i2c-0/0-004c
| |-- 0-004d -> ../../../../devices/legacy/i2c-0/0-004d
| |-- 0-004e -> ../../../../devices/legacy/i2c-0/0-004e
| |-- 0-004f -> ../../../../devices/legacy/i2c-0/0-004f
'-- drivers
    |-- i2c_adapter
    '--- lm75
        |-- 0-0048 -> ../../../../devices/legacy/i2c-0/0-0048
        |-- 0-0049 -> ../../../../devices/legacy/i2c-0/0-0049
        |-- 0-004a -> ../../../../devices/legacy/i2c-0/0-004a
        |-- 0-004b -> ../../../../devices/legacy/i2c-0/0-004b
        |-- 0-004c -> ../../../../devices/legacy/i2c-0/0-004c
        |-- 0-004d -> ../../../../devices/legacy/i2c-0/0-004d
        |-- 0-004e -> ../../../../devices/legacy/i2c-0/0-004e
        '--- 0-004f -> ../../../../devices/legacy/i2c-0/0-004f
```

在 Linux 内核源代码中的 drivers 目录下包含一个 i2c 目录，而在 i2c 目录下又包含如下文件和文件夹。

(1) i2c-core.c。

这个文件实现了 I²C 核心的功能以及 /proc/bus/i2c* 接口。

(2) i2c-dev.c。

实现了 I²C 适配器设备文件的功能，每一个 I²C 适配器都被分配一个设备。通过适配器访问设备时的主设备号都为 89，次设备号为 0~255。应用程序通过 “i2c-%d” (i2c-0, i2c-1, ..., i2c-10, ...) 文件名并使用文件操作接口 open()、write()、read()、ioctl() 和 close() 等来访问这个设备。

i2c-dev.c 并没有针对特定的设备而设计，只是提供了通用的 read()、write() 和 ioctl() 等接口，应用层可以借用这些接口访问挂载在适配器上的 I²C 设备的存储空间或寄存器，并控制 I²C 设备的工作方式。

(3) chips 文件夹。

这个目录中包含了一些特定的 I²C 设备驱动，如 Dallas 公司的 DS1337 实时钟芯片、EPSON 公司的 RTC8564 实时钟芯片和 I²C 接口的 EEPROM 驱动等。

在具体的 I²C 设备驱动中，调用的都是 I²C 核心提供的 API，因此，这使得具体的 I²C 设备驱动不依赖于 CPU 的类型和 I²C 适配器的硬件特性。

(4) busses 文件夹。

这个文件中包含了一些 I²C 总线的驱动，如针对 S3C2410、S3C2440 和 S3C6410 等处理器的 I²C 控制器驱动为 i2c-s3c2410.c。

(5) algos 文件夹。

实现了一些 I²C 总线适配器的 algorithm。



此外, 内核中的 `i2c.h` 这个头文件对 `i2c_driver`、`i2c_client`、`i2c_adapter` 和 `i2c_algorithm` 这 4 个数据结构进行了定义。理解这 4 个结构体的作用十分关键, 代码清单 15.1、15.2、15.3、15.4 分别给出了它们的定义。

代码清单 15.1 `i2c_adapter` 结构体

```
1 struct i2c_adapter {
2     struct module *owner; /*所属模块*/
3     unsigned int id; /*algorithm 的类型, 定义于 i2c-id.h, 以 I2C_ALGO_开始*/
4     unsigned int class;
5     struct i2c_algorithm *algo; /*总线通信方法结构体指针*/
6     void *algo_data; /* algorithm 数据 */
7     int (*client_register)(struct i2c_client *); /*client 注册时调用*/
8     int (*client_unregister)(struct i2c_client *); /*client 注销时调用*/
9     u8 level;
10    struct semaphore bus_lock; /*控制并发访问的自旋锁*/
11    struct semaphore clist_lock;
12    int timeout;
13    int retries; /*重试次数*/
14    struct device dev; /* 适配器设备 */
15    struct class_device class_dev; /* 类设备 */
16    int nr;
17    struct list_head clien; /* client 链表头*/
18    struct list_head list;
19    char name[48]; /*适配器名称*/
20    struct completion dev_released; /*用于同步*/
21};
```

代码清单 15.2 `i2c_algorithm` 结构体

```
1 struct i2c_algorithm {
2     int (*master_xfer)(struct i2c_adapter *adap, struct i2c_msg *msgs,
3         int num); /*i2c 传输函数指针*/
4     int (*smbus_xfer)(struct i2c_adapter *adap, u16 addr, /*smbus 传输函数指针*/
5         unsigned short flags, char read_write,
6         u8 command, int size, union i2c_smbus_data * data);
7     u32 (*functionality)(struct i2c_adapter *); /*返回适配器支持的功能*/
8 };
```

上述代码第 4 行对应为 SMBus 传输函数指针, SMBus 大部分基于 I²C 总线规范, SMBus 不需增加额外引脚。与 I²C 总线相比, SMBus 增加了一些新的功能特性, 在访问时序也有一定的差异。

代码清单 15.3 `i2c_driver` 结构体

```
1 struct i2c_driver {
2     int id;
3     unsigned int class;
4     int (*attach_adapter)(struct i2c_adapter *); /*依附 i2c_adapter 函数指针 */
5     int (*detach_adapter)(struct i2c_adapter *); /*脱离 i2c_adapter 函数指针*/
6     int (*detach_client)(struct i2c_client *); /*i2c client 脱离函数指针*/
```

```

7  int (*probe)(struct i2c_client *, const struct i2c_device_id *);
8  int (*remove)(struct i2c_client *);
9  void (*shutdown)(struct i2c_client *);
10 int (*suspend)(struct i2c_client *, pm_message_t mesg);
11 int (*resume)(struct i2c_client *);
12 int (*command)(struct i2c_client *client, unsigned int cmd, void *arg);
13 struct device_driver driver;
14 const struct i2c_device_id *id_table; /* 该驱动所支持的设备 ID 表 */
15 int (*detect)(struct i2c_client *, int kind, struct i2c_board_info *);
16 const struct i2c_client_address_data *address_data;
17 struct list_head clients;
18 };

```

代码清单 15.4 i2c_client 结构体

```

1 struct i2c_client {
2     unsigned int flags; /* 标志 */
3     unsigned short addr; /* 低 7 位为芯片地址 */
4     char name[I2C_NAME_SIZE]; /* 设备名称 */
5     struct i2c_adapter *adapter; /* 依附的 i2c_adapter */
6     struct i2c_driver *driver; /* 依附的 i2c_driver */
7     struct device dev; /* 设备结构体 */
8     int irq; /* 设备使用的中断号 */
9     struct list_head list; /* 链表头 */
10    struct completion released; /* 用于同步 */
11 };

```

下面分析 i2c_driver、i2c_client、i2c_adapter 和 i2c_algorithm 这 4 个数据结构的作用及其盘根错节的关系。

(1) i2c_adapter 与 i2c_algorithm。

i2c_adapter 对应于物理上的一个适配器，而 i2c_algorithm 对应一套通信方法。一个 I²C 适配器需要 i2c_algorithm 中提供的通信函数来控制适配器上产生特定的访问周期。缺少 i2c_algorithm 的 i2c_adapter 什么也做不了，因此 i2c_adapter 中包含其使用的 i2c_algorithm 的指针。

i2c_algorithm 中的关键函数 master_xfer() 用于产生 I²C 访问周期需要的信号，以 i2c_msg（即 I²C 消息）为单位。i2c_msg 结构体也非常关键，代码清单 15.5 给出了它的定义。

代码清单 15.5 i2c_msg 结构体

```

1 struct i2c_msg {
2     __u16 addr; /* 设备地址 */
3     __u16 flags; /* 标志 */
4     __u16 len; /* 消息长度 */
5     __u8 *buf; /* 消息数据 */
6 };

```

(2) i2c_driver 与 i2c_client。

i2c_driver 对应一套驱动方法，其主要成员函数是 probe()、remove()、suspend()、resume() 等，另外 id_table 是该驱动所支持的 I²C 设备的 ID 表。i2c_client 对应于真实的物理设备，每个 I²C 设备都需要一个 i2c_client 来描述。i2c_driver 与 i2c_client 的关系是一对多，一个 i2c_driver 上可以支持多个同等类型的 i2c_client。



i2c_client 信息通常在 BSP 的板文件中通过 i2c_board_info 填充, 如下面代码就定义了一个 I²C 设备 ID 为 “ad7142_joystick”、地址为 0x2C、中断号位 IRQ_PF5 的 i2c_client:

```
static struct i2c_board_info __initdata xxx_i2c_board_info[] = {
    #if defined(CONFIG_JOYSTICK_AD7142) || defined(CONFIG_JOYSTICK_AD7142_MODULE)
    {
        I2C_BOARD_INFO("ad7142_joystick", 0x2C),
        .irq = IRQ_PF5,
    },
    ...
}
```

在 I²C 总线驱动 i2c_bus_type 的 match() 函数 i2c_device_match() 中, 会调用 i2c_match_id() 函数匹配板文件中定义的 ID 和 i2c_driver 所支持的 ID 表。

(3) i2c_adapter 与 i2c_client。

i2c_adapter 与 i2c_client 的关系与 I²C 硬件体系中适配器和设备的关系一致, 即 i2c_client 依附于 i2c_adapter。由于一个适配器上可以连接多个 I²C 设备, 所以一个 i2c_adapter 也可以被多个 i2c_client 依附, i2c_adapter 中包括依附于它的 i2c_client 的链表。

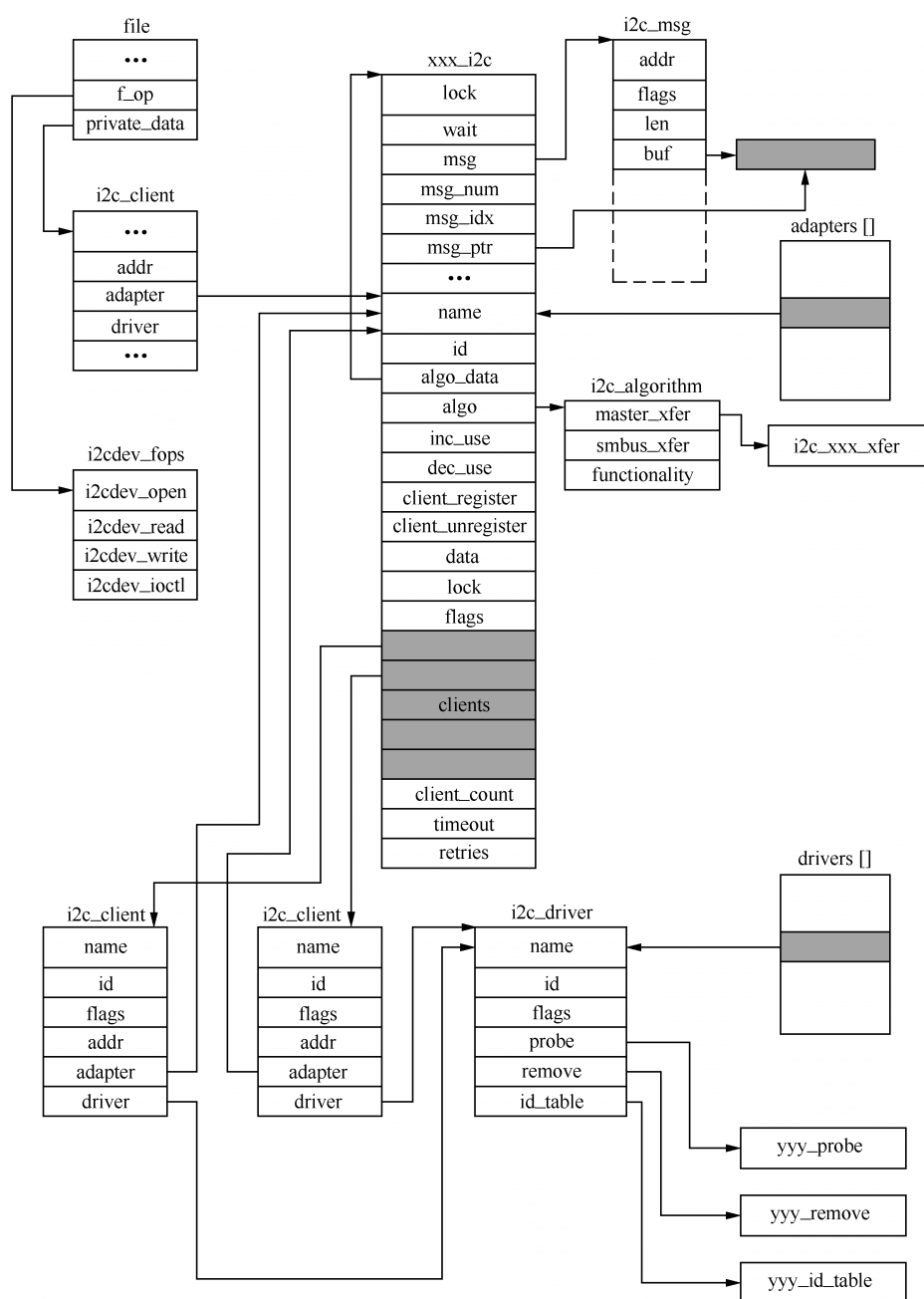
假设 I²C 总线适配器 xxx 上有两个使用相同驱动程序的 yyy I²C 设备, 在打开该 I²C 总线的设备结点后相关数据结构之间的逻辑组织关系将如图 15.2 所示。

从上面的分析可知, 虽然 I²C 硬件体系结构比较简单, 但是 I²C 体系结构在 Linux 中的实现却相当复杂。当工程师拿到实际的电路板, 面对复杂的 Linux I²C 子系统, 应该如何下手写驱动呢? 究竟有哪些是需要亲自做的, 哪些是内核已经提供的呢? 理清这个问题非常有意义, 可以使我们对具体问题时迅速地抓住重点。

一方面, 适配器驱动可能是 Linux 内核本身还不包含的; 另一方面, 挂接在适配器上的具体设备驱动可能也是 Linux 内核还不包含的。因此, 工程师要实现的主要工作如下。

- 提供 I²C 适配器的硬件驱动, 探测、初始化 I²C 适配器 (如申请 I²C 的 I/O 地址和中断号)、驱动 CPU 控制的 I²C 适配器从硬件上产生各种信号以及处理 I²C 中断等。
- 提供 I²C 适配器的 algorithm, 用具体适配器的 xxx_xfer() 函数填充 i2c_algorithm 的 master_xfer 指针, 并把 i2c_algorithm 指针赋值给 i2c_adapter 的 algo 指针。
- 实现 I²C 设备驱动中的 i2c_driver 接口, 用具体设备 yyy 的 yyy_probe()、yyy_remove()、yyy_suspend()、yyy_resume() 函数指针和 i2c_device_id 设备 ID 表赋值给 i2c_driver 的 probe、remove、suspend、resume 和 id_table 指针。
- 实现 I²C 设备所对应类型的具体驱动, i2c_driver 只是实现设备与总线的挂接, 而挂接在总线上的设备则是千差万别。例如, 如果是字符设备, 就实现文件操作接口, 即实现具体设备 yyy 的 yyy_read()、yyy_write() 和 yyy_ioctl() 函数等; 如果是声卡, 就实现 ALSA 驱动。

上述工作中前两个属于 I²C 总线驱动, 后两个属于 I²C 设备驱动, 做完这些工作, 系统会增加两个内核模块。15.3~15.4 节将详细分析这些工作的实施方法, 给出设计模板, 而 15.5~15.6 节将给出两个具体的实例。

图 15.2 I²C 驱动的各数据结构的关系

15.2 Linux I²C 核心

I²C 核心（drivers/i2c/i2c-core.c）中提供了一组不依赖于硬件平台的接口函数，这个文件一般



不需要被工程师修改,但是理解其中的主要函数非常关键,因为 I²C 总线驱动和设备驱动之间依赖于 I²C 核心作为纽带。I²C 核心中的主要函数如下。

(1) 增加/删除 i2c_adapter。

```
int i2c_add_adapter(struct i2c_adapter *adap);
int i2c_del_adapter(struct i2c_adapter *adap);
```

(2) 增加/删除 i2c_driver。

```
int i2c_register_driver(struct module *owner, struct i2c_driver *driver);
int i2c_del_driver(struct i2c_driver *driver);
inline int i2c_add_driver(struct i2c_driver *driver);
```

(3) i2c_client 依附/脱离。

```
int i2c_attach_client(struct i2c_client *client);
int i2c_detach_client(struct i2c_client *client);
```

当一个具体的 client 被检测到并被关联的时候,设备和 sysfs 文件将被注册。相反地,在 client 被取消关联的时候,sysfs 文件和设备也被注销,如代码清单 15.6 所示。

代码清单 15.6 I²C 核心的 client attach/detach 函数

```
1 int i2c_attach_client(struct i2c_client *client)
2 {
3     ...
4     device_register(&client->dev);
5     ...
6 }
7
8 int i2c_detach_client(struct i2c_client *client)
9 {
10    ...
11    device_unregister(&client->dev);
12    ...
13 }
```

(4) I²C 传输、发送和接收。

```
int i2c_transfer(struct i2c_adapter * adap, struct i2c_msg *msgs, int num);
int i2c_master_send(struct i2c_client *client, const char *buf, int count);
int i2c_master_recv(struct i2c_client *client, char *buf, int count);
```

i2c_transfer()函数用于进行 I²C 适配器和 I²C 设备之间的一组消息交互, i2c_master_send()函数和 i2c_master_recv()函数内部会调用 i2c_transfer()函数分别完成一条写消息和一条读消息,如代码清单 15.7、代码清单 15.8 所示。

代码清单 15.7 I²C 核心的 i2c_master_send 函数

```
1 int i2c_master_send(struct i2c_client *client, const char *buf, int count)
2 {
3     int ret;
4     struct i2c_adapter *adap=client->adapter;
5     struct i2c_msg msg;
6     /*构造一个写消息*/
7     msg.addr = client->addr;
8     msg.flags = client->flags & I2C_M_TEN;
9     msg.len = count;
10    msg.buf = (char *)buf;
11    /*传输消息*/
```

```

12 ret = i2c_transfer(adap, &msg, 1);
13
14 return (ret == 1) ? count : ret;
15 }

```

代码清单 15.8 I²C 核心的 i2c_master_recv 函数

```

1 int i2c_master_recv(struct i2c_client *client, char *buf, int count)
2 {
3     struct i2c_adapter *adap=client->adapter;
4     struct i2c_msg msg;
5     int ret;
6     /*构造一个读消息*/
7     msg.addr = client->addr;
8     msg.flags = client->flags & I2C_M_TEN;
9     msg.flags |= I2C_M_RD;
10    msg.len = count;
11    msg.buf = buf;
12    /*传输消息*/
13    ret = i2c_transfer(adap, &msg, 1);
14
15    /* 成功 (1 条消息被处理), 返回读的字节数 */
16    return (ret == 1) ? count : ret;
17 }

```

i2c_transfer()函数本身不具备驱动适配器物理硬件完成消息交互的能力，它只是寻找到 i2c_adapter 对应的 i2c_algorithm，并使用 i2c_algorithm 的 master_xfer()函数真正驱动硬件流程，如代码清单 15.9 所示。

代码清单 15.9 I²C 核心的 i2c_transfer 函数

```

1 int i2c_transfer(struct i2c_adapter * adap, struct i2c_msg *msgs, int num)
2 {
3     int ret;
4
5     if (adap->algo->master_xfer) {
6         ...
7         ret = adap->algo->master_xfer(adap, msgs, num); /* 消息传输 */
8         ...
9         return ret;
10    } else {
11        dev_dbg(&adap->dev, "I2C level transfers not supported\n");
12        return -ENOSYS;
13    }
14 }

```

15.3 Linux I²C 总线驱动

15.3.1 I²C 适配器驱动加载与卸载

I²C 总线驱动模块的加载函数要完成两个工作。

- 初始化 I²C 适配器所使用的硬件资源，如申请 I/O 地址、中断号等。



- 通过 `i2c_add_adapter()` 添加 `i2c_adapter` 的数据结构, 当然这个 `i2c_adapter` 数据结构的成员已经被 `xxx` 适配器的相应函数指针所初始化。

I²C 总线驱动模块的卸载函数要完成的工作与加载函数相反。

- 释放 I²C 适配器所使用的硬件资源, 如释放 I/O 地址、中断号等。
- 通过 `i2c_del_adapter()` 删除 `i2c_adapter` 的数据结构。

代码清单 15.10 所示为 I²C 适配器驱动模块加载和卸载函数的模板。

代码清单 15.10 I²C 总线驱动模块加载和卸载函数模板

```
1 static int __init i2c_adapter_xxx_init(void)
2 {
3     xxx_adpater_hw_init();
4     i2c_add_adapter(&xxx_adapter);
5 }
6
7 static void __exit i2c_adapter_xxx_exit(void)
8 {
9     xxx_adpater_hw_free();
10    i2c_del_adapter(&xxx_adapter);
11 }
```

上述代码中 `xxx_adpater_hw_init()` 和 `xxx_adpater_hw_free()` 函数的实现都与具体的 CPU 和 I²C 适配器硬件直接相关。

15.3.2 I²C 总线通信方法

我们需要为特定的 I²C 适配器实现其通信方法, 主要实现 `i2c_algorithm` 的 `master_xfer()` 函数和 `functionality()` 函数。

`functionality()` 函数非常简单, 用于返回 `algorithm` 所支持的通信协议, 如 `I2C_FUNC_I2C`、`I2C_FUNC_10BIT_ADDR`、`I2C_FUNC_SMBUS_READ_BYTE`、`I2C_FUNC_SMBUS_WRITE_BYTE` 等。

`master_xfer()` 函数在 I²C 适配器上完成传递给它的 `i2c_msg` 数组中的每个 I²C 消息, 代码清单 15.11 所示为 `xxx` 设备的 `master_xfer()` 函数模板。

代码清单 15.11 I²C 总线驱动 `master_xfer` 函数模板

```
1 static int i2c_adapter_xxx_xfer(struct i2c_adapter *adap, struct i2c_msg *msgs,
2     int num)
3 {
4     ...
5     for (i = 0; i < num; i++) {
6         i2c_adapter_xxx_start(); /*产生开始位*/
7         /*是读消息*/
8         if (msgs[i]->flags & I2C_M_RD) {
9             i2c_adapter_xxx_setaddr(msg->addr << 1); /*发送从设备读地址*/
10            i2c_adapter_xxx_wait_ack(); /*获得从设备的 ack*/
11            i2c_adapter_xxx_readbytes(msgs[i]->buf, msgs[i]->len); /*读取 msgs[i] ->len
12                长的数据到 msgs[i]->buf*/
13        } else { /*是写消息*/
14            i2c_adapter_xxx_setaddr(msg->addr << 1); /*发送从设备写地址*/
15            i2c_adapter_xxx_wait_ack(); /*获得从设备的 ack*/
16            i2c_adapter_xxx_writebytes(msgs[i]->buf, msgs[i]->len); /*读取 msgs[i] ->len
```



```
17         长的数据到 msgs[i]->buf*/
18     }
19 }
20 i2c_adapter_xxx_stop(); /*产生停止位*/
21 }
```

上述代码实际上给出了一个 master_xfer()函数处理 I²C 消息数组的流程，对于数组中的每个消息，判断消息类型，若为读消息，则赋从设备地址为 (msg->addr << 1) | 1，否则为 msg->addr << 1。对每个消息产生一个开始位，紧接着传送从设备地址，然后开始数据的发送或接收，对最后的消息还需产生一个停止位。如图 15.3 所示为整个 master_xfer()完成的时序。

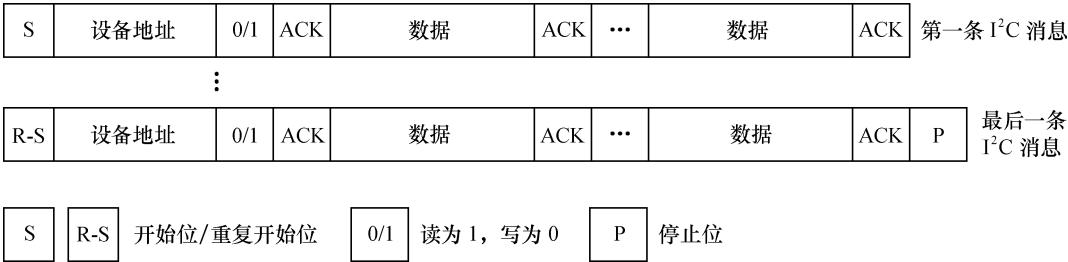


图 15.3 algorithm 中 master_xfer 的时序

master_xfer()函数模板中的 i2c_adapter_xxx_start()、i2c_adapter_xxx_setaddr()、i2c_adapter_xxx_wait_ack()、i2c_adapter_xxx_readbytes()、i2c_adapter_xxx_writebytes()和 i2c_adapter_xxx_stop()函数用于完成适配器的底层硬件操作，与 I²C 适配器和 CPU 的具体硬件直接相关，需要由工程师根据芯片的数据手册来实现。

i2c_adapter_xxx_readbytes()用于从从设备上接收一串数据，i2c_adapter_xxx_writebytes()用于向从设备写入一串数据，这两个函数的内部也会涉及 I²C 总线协议中的 ACK 应答。

master_xfer()函数的实现在形式上会很多样，即便是 Linux 内核源代码中已经给出的一些 I²C 总线驱动的 master_xfer()函数，由于由不同的组织或个人完成，风格上的差别也非常大，不一定能与模板完全对应，如 master_xfer()函数模板给出的消息处理是顺序进行的，而有的驱动以中断方式来完成这个流程（15.5 节的实例即是如此）。不管具体怎么实施，流程的本质都是不变的。因为这个流程不以驱动工程师的意志为转移，最终由 I²C 总线硬件上的通信协议决定。

多数 I²C 总线驱动会定义一个 xxx_i2c 结构体，作为 i2c_adapter 的 algo_data（类似“私有数据”），其中包含 I²C 消息数组指针、数组索引及 I²C 适配器 algorithm 访问控制用的自旋锁、等待队列等，而 master_xfer()函数完成消息数组中消息的处理也可通过对 xxx_i2c 结构体相关成员的访问来控制。代码清单 15.12 所示为 xxx_i2c 结构体的定义，与图 15.2 中的 xxx_i2c 是对应的。

代码清单 15.12 xxx_i2c 结构体模板

```
1 struct xxx_i2c {
2     spinlock_t      lock;
3     wait_queue_head_t wait;
4     struct i2c_msg   *msg;
5     unsigned int     msg_num;
```



```
6 unsigned int      msg_idx;  
7 unsigned int      msg_ptr;  
8 ...  
9 struct i2c_adapter adap;  
10 };
```

15.4 Linux I²C 设备驱动

I²C 设备驱动要使用 `i2c_driver` 和 `i2c_client` 数据结构并填充 `i2c_driver` 中的成员函数。`i2c_client` 一般被包含在设备的私有信息结构体 `yyy_data` 中, 而 `i2c_driver` 则适合被定义为全局变量并初始化, 代码清单 15.13 所示为已被初始化的 `i2c_driver`。

代码清单 15.13 已被初始化的 `i2c_driver`

```
1 static struct i2c_driver yyy_driver = {  
2     .driver = {  
3         .name = "yyy",  
4     },  
5     .probe      = yyy_probe,  
6     .remove     = yyy_remove,  
7     .id_table   = yyy_id,  
8 };
```

15.4.1 Linux I²C 设备驱动的模块加载与卸载

I²C 设备驱动的模块加载函数通用的方法是在 I²C 设备驱动的模块加载函数进行通过 I²C 核心的 `i2c_add_driver()` 函数添加 `i2c_driver` 的工作, 而在模块卸载函数中需要做相反的工作: 通过 I²C 核心的 `i2c_del_driver()` 函数删除 `i2c_driver`。代码清单 15.14 所示为 I²C 设备驱动的加载工作与卸载函数模板。

```
1 static int __init yyy_init(void)  
2 {  
3     rern i2c_add_driver(&yyy_driver);  
4 };  
4 void __exit yyy_exit(void)  
5 {  
6     i2c_del_driver(&yyy_driver);  
7 };
```

15.4.2 Linux I²C 设备驱动的数据传输

在 I²C 设备上读写数据的时序和数据通常通过 `i2c_msg` 数组组织, 最后通过 `i2c_transfer()` 函数完成, 代码清单 15.15 所示为一个读取指定偏移 `offs` 寄存器的例子。

代码清单 15.14 I²C 设备驱动数据传输范例

```
1 struct i2c_msg msg[2];  
2 /*第一条消息是写消息*/  
3 msg[0].addr = client->addr;  
4 msg[0].flags = 0;  
5 msg[0].len = 1;
```

```

6  msg[0].buf = &offs;
7  /*第二条消息是读消息*/
8  msg[1].addr = client->addr;
9  msg[1].flags = I2C_M_RD;
10 msg[1].len = sizeof(buf);
11 msg[1].buf = &buf[0];
12
13 i2c_transfer(client->adapter, msg, 2);

```

15.4.3 Linux 的 i2c-dev.c 文件分析

i2c-dev.c 文件完全可以被看作一个 I²C 设备驱动，不过，它实现的一个 i2c_client 是虚拟、临时的，随着设备文件的打开而产生，并随设备文件的关闭而撤销，并没有被添加到 i2c_adapter 的 clien 链表中。i2c-dev.c 针对每个 I²C 适配器生成一个主设备号为 89 的设备文件，实现了 i2c_driver 的成员函数以及文件操作接口，所以 i2c-dev.c 的主体是“i2c_driver 成员函数+字符设备驱动”。

i2c-dev.c 中提供 i2cdev_read()、i2cdev_write()函数来对应用户空间要使用的 read()和 write()文件操作接口，这两个函数分别调用 I²C 核心的 i2c_master_recv()和 i2c_master_send()函数来构造一条 I²C 消息并引发适配器 algorithm 通信函数的调用，完成消息的传输，对应于如图 15.4 所示的时序。但是，很遗憾，大多数稍微复杂一点 I²C 设备的读写流程并不对应于一条消息，往往需要两条甚至更多的消息来进行一次读写周期（即如图 15.5 所示的重复开始位 RepStart 模式），这种情况下，在应用层仍然调用 read()、write()文件 API 来读写 I²C 设备，将不能正确地读写。许多工程师碰到过类似的问题，往往经过相当长时间的调试都没法解决 I²C 设备的读写，连错误的原因也无法找到，显然是对 i2cdev_read()和 i2cdev_write()函数的作用有所误解。



图 15.4 i2cdev_read 和 i2cdev_write 函数对应时序



图 15.5 RepStart 模式

鉴于上述原因，i2c-dev.c 中 i2cdev_read()和 i2cdev_write()函数不具备太强的通用性，没有太大的实用价值，只能适用于非 RepStart 模式的情况。对于两条以上消息组成的读写，在用户空间需要组织 i2c_msg 消息数组并调用 I2C_RDWR IOCTL 命令。代码清单 15.15 所示为 i2cdev_ioctl()函数的框架。

代码清单 15.15 i2c-dev_c 中的 i2cdev_ioctl 函数

```

1 static int i2cdev_ioctl(struct inode *inode, struct file *file,
2     unsigned int cmd, unsigned long arg)
3 {
4     struct i2c_client *client = (struct i2c_client *)file->private_data;
5     ...
6     switch ( cmd ) {
7     case I2C_SLAVE:
8     case I2C_SLAVE_FORCE:

```



```
9      ... /*设置从设备地址*/
10 case I2C_TENBIT:
11     ...
12 case I2C_PEC:
13     ...
14 case I2C_FUNCS:
15     ...
16 case I2C_RDWR:
17     return i2cdev_ioctl_rdrw(client, arg);
18 case I2C_SMBUS:
19     ...
20 case I2C_RETRIES:
21     ...
22 case I2C_TIMEOUT:
23     ...
24 default:
25     return i2c_control(client, cmd, arg);
26 }
27 return 0;
28 }
```

常用的 IOCTL 包括 I2C_SLAVE (设置从设备地址)、I2C_RETRIES (没有收到设备 ACK 情况下的重试次数, 默认为 1)、I2C_TIMEOUT (超时) 以及 I2C_RDWR。

代码清单 15.16 和代码清单 15.17 所示为直接通过 read()、write() 接口和 O_RDWR IOCTL 读写 I²C 设备的例子。

代码清单 15.16 直接通过 read()/write() 读写 I²C 设备

```
1 #include <stdio.h>
2 #include <linux/types.h>
3 #include <fcntl.h>
4 #include <unistd.h>
5 #include <stdlib.h>
6 #include <sys/types.h>
7 #include <sys/ioctl.h>
8 #include <linux/i2c.h>
9 #include <linux/i2c-dev.h>
10
11 int main(int argc, char **argv)
12 {
13     unsigned int fd;
14     unsigned short mem_addr;
15     unsigned short size;
16     unsigned short idx;
17     #define BUFF_SIZE 32
18     char buf[BUFF_SIZE];
19     char cswap;
20     union
21     {
22         unsigned short addr;
23         char bytes[2];
24     } tmp;
25
26     if (argc < 3) {
27         printf("Use: %s /dev/i2c-x mem_addr size\n", argv[0]);
```

```

28     return 0;
29 }
30 sscanf(argv[2], "%d", &mem_addr);
31 sscanf(argv[3], "%d", &size);
32
33 if (size > BUFF_SIZE)
34     size = BUFF_SIZE;
35
36 fd = open(argv[1], O_RDWR);
37
38 if (!fd) {
39     printf("Error on opening the device file\n");
40     return 0;
41 }
42
43 ioctl(fd, I2C_SLAVE, 0x50); /* 设置 EEPROM 地址 */
44 ioctl(fd, I2C_TIMEOUT, 1); /* 设置超时 */
45 ioctl(fd, I2C_RETRIES, 1); /* 设置重试次数 */
46
47 for (idx = 0; idx < size; ++idx, ++mem_addr) {
48     tmp.addr = mem_addr;
49     cswap = tmp.bytes[0];
50     tmp.bytes[0] = tmp.bytes[1];
51     tmp.bytes[1] = cswap;
52     write(fd, &tmp.addr, 2);
53     read(fd, &buf[idx], 1);
54 }
55 buf[size] = 0;
56 close(fd);
57 printf("Read %d char: %s\n", size, buf);
58 return 0;
59 }

```

代码清单 15.17 通过 O_RDWR IOCTL 读写 I²C 设备

```

1 #include <stdio.h>
2 #include <linux/types.h>
3 #include <fcntl.h>
4 #include <unistd.h>
5 #include <stdlib.h>
6 #include <sys/types.h>
7 #include <sys/ioctl.h>
8 #include <errno.h>
9 #include <assert.h>
10 #include <string.h>
11 #include <linux/i2c.h>
12 #include <linux/i2c-dev.h>
13
14 int main(int argc, char **argv)
15 {
16     struct i2c_rdwr_ioctl_data work_queue;
17     unsigned int idx;
18     unsigned int fd;
19     unsigned int slave_address, reg_address;
20     unsigned char val;

```



```
21 int i;
22 int ret;
23
24 if (argc < 4) {
25     printf("Usage:\n%s /dev/i2c-x start_addr reg_addr\n", argv[0]);
26     return 0;
27 }
28
29 fd = open(argv[1], O_RDWR);
30
31 if (!fd) {
32     printf("Error on opening the device file\n");
33     return 0;
34 }
35 sscanf(argv[2], "%x", &slave_address);
36 sscanf(argv[3], "%x", &reg_address);
37
38 work_queue.nmsgs = 2; /* 消息数量 */
39 work_queue.msgs = (struct i2c_msg*)malloc(work_queue.nmsgs * sizeof(struct
40     i2c_msg));
41 if (!work_queue.msgs) {
42     printf("Memory alloc error\n");
43     close(fd);
44     return 0;
45 }
46
47 ioctl(fd, I2C_TIMEOUT, 2); /* 设置超时 */
48 ioctl(fd, I2C_RETRIES, 1); /* 设置重试次数 */
49
50 for (i = reg_address; i < reg_address + 16; i++) {
51     val = i;
52     (work_queue.msgs[0]).len = 1;
53     (work_queue.msgs[0]).addr = slave_address;
54     (work_queue.msgs[0]).buf = &val;
55
56     (work_queue.msgs[1]).len = 1;
57     (work_queue.msgs[1]).flags = I2C_M_RD;
58     (work_queue.msgs[1]).addr = slave_address;
59     (work_queue.msgs[1]).buf = &val;
60
61     ret = ioctl(fd, I2C_RDWR, (unsigned long) &work_queue);
62     if (ret < 0)
63         printf("Error during I2C_RDWR ioctl with error code: %d\n", ret);
64     else
65         printf("reg:%02x val:%02x\n", i, val);
66 }
67 close(fd);
68 return ;
69 }
```

该程序位于虚拟机的/home/lihacker/develop/svn/ldd6410-read-only/tes/i2c/i2c-base-test 目录, 使用该工具可指定读取某 I²C 控制器上某 I²C 从设备的某寄存器, 如读 I²C 控制器 0 上的地址为 0x18 的从设备, 从寄存器 0x20 开始读:

```
# i2c-test /dev/i2c-0 0x18 0x20
reg:20 val:07
```

```

reg:21 val:00
reg:22 val:00
reg:23 val:00
reg:24 val:00
reg:25 val:00
reg:26 val:00
reg:27 val:00
reg:28 val:00
reg:29 val:00
reg:2a val:00
reg:2b val:00
reg:2c val:00
reg:2d val:00
reg:2e val:00
reg:2f val:00

```

15.5 S3C6410 I²C 总线驱动实例

15.5.1 S3C6410 I²C 控制器硬件描述

S3C6410 处理器内部集成了一个 I²C 控制器，通过 4 个寄存器就可方便地对其进行控制，这 4 个寄存器如下。

- IICCON: I²C 控制寄存器。
- IICSTAT: I²C 状态寄存器。
- IICDS: I²C 收发数据移位寄存器。
- IICADD: I²C 地址寄存器。

S3C6410 处理器内部集成的 I²C 控制器可支持主、从两种模式，我们主要使用其主模式。通过对 IICCON、IICDS 和 IICADD 寄存器的操作，可在 I²C 总线上产生开始位、停止位、数据和地址，而传输的状态则通过 IICSTAT 寄存器获取。

15.5.2 S3C6410 I²C 总线驱动总体分析

S3C6410 的 I²C 总线驱动 drivers/i2c/busses/i2c-s3c2410.c 支持 S3C24XX、S3C64XX、S5PC1XX 和 S5P64XX 处理器，在我们使用的 2.6.28.6 内核版本中，其名称仍然叫 2410，显然是历史原因引起的。它主要完成以下工作。

(1) 设计对应于 i2c_adapter_xxx_init() 模板的 S3C6410 的模块加载函数和对应于 i2c_adapter_xxx_exit() 函数模板的模块卸载函数。

(2) 设计对应于 i2c_adapter_xxx_xfer() 模板的 S3C6410 适配器的通信方法函数。

针对 S3C24XX、S3C64XX、S5PC1XX 和 S5P64XX 处理器，functionality() 函数 s3c24xx_i2c_func() 只需简单地返回 I2C_FUNC_I2C|I2C_FUNC_SMBUS_EMUL|I2C_FUNC_PROTOCOL_MANGLING 表明其支持的功能。

图 15.6 给出了 S3C6410 驱动中的主要函数与 15.3 节模板函数的对应关系，由于实现通信方



法的方式不一样, 模板的一个函数可能对应于 S3C6410 I²C 总线驱动的多个函数。

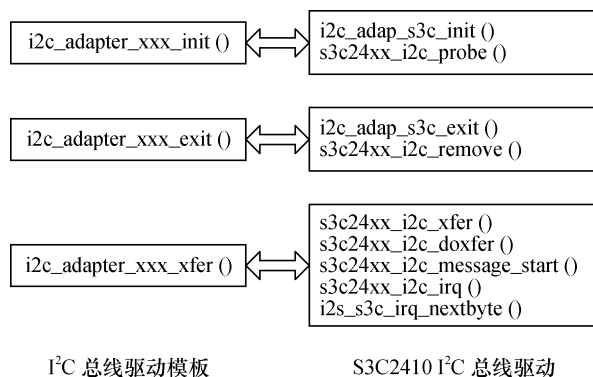


图 15.6 I²C 总线驱动模板与 S3C6410 I²C 总线驱动的映射

15.5.3 S3C6410 I²C 适配器驱动的模式加载与卸载

I²C 适配器驱动被作为一个单独的模块加载进内核, 在模块的加载和卸载函数中, 只需注册和注销一个 platform_driver 结构体, 如代码清单 15.18 所示。

代码清单 15.18 S3C6410 I²C 总线驱动的模式加载与卸载

```
1 static int __init i2c_adap_s3c_init(void)
2 {
3     int ret;
4
5     ret = platform_driver_register(&s3c2410_i2c_driver);
6     if (ret == 0) {
7         ret = platform_driver_register(&s3c2440_i2c_driver);
8         if (ret)
9             platform_driver_unregister(&s3c2410_i2c_driver);
10    }
11
12    return ret;
13 }
14
15 static void __exit i2c_adap_s3c_exit(void)
16 {
17     platform_driver_unregister(&s3c2410_i2c_driver);
18     platform_driver_unregister(&s3c2440_i2c_driver);
19 }
20 module_init(i2c_adap_s3c_init);
21 module_exit(i2c_adap_s3c_exit);
```

platform_driver 结构体包含了具体适配器的 probe()函数、remove()函数、resume()函数指针等信息, 它需要被定义和赋值, 如代码清单 15.19 所示。

代码清单 15.19 platform_driver 结构体

```
1 static struct platform_driver s3c2410_i2c_driver = {
2     .probe      = s3c24xx_i2c_probe,
3     .remove     = s3c24xx_i2c_remove,
4     .resume     = s3c24xx_i2c_resume,
```



```

5  .driver      = {
6      .owner    = THIS_MODULE,
7      .name     = "s3c2410-i2c",
8  },
9  };

```

当通过 Linux 内核源代码/drivers/base/platform.c 文件中定义 platform_driver_unregister() 函数注册 platform_driver 结构体时, 其中 probe 指针指向的 s3c24xx_i2c_probe() 函数将被调用, 以初始化适配器硬件, 如代码清单 15.20 所示。

代码清单 15.20 S3C6410 I²C 总线驱动中的 s3c24xx_i2c_probe 函数

```

1  static int s3c24xx_i2c_probe(struct platform_device *pdev)
2  {
3      struct s3c24xx_i2c *i2c;
4      struct s3c2410_platform_i2c *pdata;
5      struct resource *res;
6      int ret;
7
8      pdata = pdev->dev.platform_data;
9      if (!pdata) {
10         dev_err(&pdev->dev, "no platform data\n");
11         return -EINVAL;
12     }
13
14     i2c = kzalloc(sizeof(struct s3c24xx_i2c), GFP_KERNEL);
15     if (!i2c) {
16         dev_err(&pdev->dev, "no memory for state\n");
17         return -ENOMEM;
18     }
19
20     strcpy(i2c->adap.name, "s3c2410-i2c", sizeof(i2c->adap.name));
21     i2c->adap.owner    = THIS_MODULE;
22     i2c->adap.algo     = &s3c24xx_i2c_algorithm;
23     i2c->adap.retries  = 2;
24     i2c->adap.class    = I2C_CLASS_HWMON | I2C_CLASS_SPD;
25     i2c->tx_sep        = 50;
26
27     spin_lock_init(&i2c->lock);
28     init_waitqueue_head(&i2c->wait);
29
30     /* 发现时钟并使能它 */
31
32     i2c->dev = &pdev->dev;
33     i2c->clk = clk_get(&pdev->dev, "i2c");
34     if (IS_ERR(i2c->clk)) {
35         dev_err(&pdev->dev, "cannot get clock\n");
36         ret = -ENOENT;
37         goto err_noclk;
38     }
39
40     dev_dbg(&pdev->dev, "clock source %p\n", i2c->clk);
41
42     clk_enable(i2c->clk);
43

```



```
44 /* 映射寄存器 */
45
46 res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
47 if (res == NULL) {
48     dev_err(&pdev->dev, "cannot find IO resource\n");
49     ret = -ENOENT;
50     goto err_clk;
51 }
52
53 i2c->ioarea = request_mem_region(res->start, (res->end-res->start)+1,
54     pdev->name);
55
56 if (i2c->ioarea == NULL) {
57     dev_err(&pdev->dev, "cannot request IO\n");
58     ret = -ENXIO;
59     goto err_clk;
60 }
61
62 i2c->regs = ioremap(res->start, (res->end-res->start)+1);
63
64 if (i2c->regs == NULL) {
65     dev_err(&pdev->dev, "cannot map IO\n");
66     ret = -ENXIO;
67     goto err_ioarea;
68 }
69
70 dev_dbg(&pdev->dev, "registers %p (%p, %p)\n",
71     i2c->regs, i2c->ioarea, res);
72
73 /* 设置 i2c 核心需要的信息 */
74
75 i2c->adap.algo_data = i2c;
76 i2c->adap.dev.parent = &pdev->dev;
77
78 /* initialise the i2c controller */
79
80 ret = s3c24xx_i2c_init(i2c);
81 if (ret != 0)
82     goto err_iomap;
83
84 /* 申请中断
85  */
86
87 i2c->irq = ret = platform_get_irq(pdev, 0);
88 if (ret <= 0) {
89     dev_err(&pdev->dev, "cannt find IRQ\n");
90     goto err_iomap;
91 }
92
93 ret = request_irq(i2c->irq, s3c24xx_i2c_irq, IRQF_DISABLED,
94     dev_name(&pdev->dev), i2c);
95
96 if (ret != 0) {
97     dev_err(&pdev->dev, "cannot claim IRQ %d\n", i2c->irq);
98     goto err_iomap;
```

```

99 }
100
101 ret = s3c24xx_i2c_register_cpufreq(i2c);
102 if (ret < 0) {
103     dev_err(&pdev->dev, "failed to register cpufreq notifier\n");
104     goto err_irq;
105 }
106
107 /* bus_num 是 platform 数据
108 */
109
110 i2c->adap.nr = pdata->bus_num;
111
112 ret = i2c_add_numbered_adapter(&i2c->adap);
113 if (ret < 0) {
114     dev_err(&pdev->dev, "failed to add bus to i2c core\n");
115     goto err_cpufreq;
116 }
117
118 platform_set_drvdata(pdev, i2c);
119
120 dev_info(&pdev->dev, "%s: S3C I2C adapter\n", i2c->adap.dev.bus_id);
121 return 0;
122
123 ...
124 }

```

上述代码中的主体工作是使能硬件并申请 I²C 适配器使用的 I/O 地址、中断号等，在这些工作都完成无误后，通过 I²C 核心提供的 `i2c_add_adapter()` 函数添加这个适配器。当处理器包含多个 I²C 控制器时，我们通过板文件定义的 `platform` 数据中的 `bus_num` 进行区分。

与 `s3c24xx_i2c_probe()` 函数完成相反功能的函数是 `s3c24xx_i2c_remove()` 函数，它在适配器模块卸载函数调用 `platform_driver_unregister()` 函数时通过 `platform_driver` 的 `remove` 指针方式被调用。`s3c24xx_i2c_remove()` 的设计模板如代码清单 15.21 所示。

代码清单 15.21 S3C6410 I²C 总线驱动中的 `s3c24xx_i2c_remove` 函数

```

1 static int s3c24xx_i2c_remove(struct platform_device *pdev)
2 {
3     struct s3c24xx_i2c *i2c = platform_get_drvdata(pdev);
4
5     s3c24xx_i2c_deregister_cpufreq(i2c);
6
7     i2c_del_adapter(&i2c->adap);
8     free_irq(i2c->irq, i2c);
9
10    clk_disable(i2c->clk);
11    clk_put(i2c->clk);
12
13    iounmap(i2c->regs);
14
15    release_resource(i2c->ioarea);
16    kfree(i2c->ioarea);
17    kfree(i2c);
18
19    return 0;
20 }

```



代码清单 15.20 和代码清单 15.21 中用到的 `s3c24xx_i2c` 结构体进行适配器所有信息的封装, 类似于私有信息结构体, 它与代码清单 15.12 所示的 `xxx_i2c` 结构体模板对应。代码清单 15.22 所示为 `s3c24xx_i2c` 结构体的定义。

代码清单 15.22 `s3c24xx_i2c` 结构体

```
1 struct s3c24xx_i2c {
2     spinlock_t      lock;
3     wait_queue_head_t wait;
4     unsigned int     suspended:1;
5
6     struct i2c_msg    *msg;
7     unsigned int      msg_num;
8     unsigned int      msg_idx;
9     unsigned int      msg_ptr;
10
11    unsigned int       tx_setup;
12    unsigned int       irq;
13
14    enum s3c24xx_i2c_state state;
15    unsigned long      clkrate;
16
17    void __iomem       *regs;
18    struct clk          *clk;
19    struct device       *dev;
20    struct resource     *ioarea;
21    struct i2c_adapter  adap;
22
23    #ifdef CONFIG_CPU_FREQ
24    struct notifier_block freq_transition;
25    #endif
26 };
```

15.5.4 S3C6410 I²C 总线通信方法

由代码清单 15.20 的第 22 行可以看出, I²C 适配器对应的 `i2c_algorithm` 结构体实例为 `s3c24xx_i2c_algorithm`, 代码清单 15.23 所示为 `s3c24xx_i2c_algorithm` 的定义。

代码清单 15.23 S3C6410 的 `i2c_algorithm` 结构体

```
1 static struct i2c_algorithm s3c24xx_i2c_algorithm = {
2     .master_xfer      = s3c24xx_i2c_xfer,
3     .functionality     = s3c24xx_i2c_func,
4 };
```

上述代码第一行指定了 S3C6410 I²C 总线通信传输函数 `s3c24xx_i2c_xfer()`, 这个函数非常关键, 所有 I²C 总线上对设备的访问最终应该由它来完成, 代码清单 15.24 所示为这个重要函数以及其依赖的 `s3c24xx_i2c_doxfer()` 函数和 `s3c24xx_i2c_message_start()` 函数的源代码。

代码清单 15.24 S3C6410 I²C 总线驱动的 `master_xfer` 函数

```
1 static int s3c24xx_i2c_xfer(struct i2c_adapter *adap,
2     struct i2c_msg *msgs, int num)
3 {
4     struct s3c24xx_i2c *i2c = (struct s3c24xx_i2c *)adap->algo_data;
```

```

5   int retry;
6   int ret;
7
8   for (retry = 0; retry < adap->retries; retry++) {
9
10      ret = s3c24xx_i2c_doxfer(i2c, msgs, num);
11
12      if (ret != -EAGAIN)
13          return ret;
14
15      dev_dbg(i2c->dev, "Retrying transmission (%d)\n", retry);
16
17      udelay(100);
18  }
19
20  return -EREMOTEIO;
21  }
22
23  static int s3c24xx_i2c_doxfer(struct s3c24xx_i2c *i2c,
24                              struct i2c_msg *msgs, int num)
25  {
26      unsigned long timeout;
27      int ret;
28      int iicstat;
29
30      if (i2c->suspended)
31          return -EIO;
32
33      ret = s3c24xx_i2c_set_master(i2c);
34      if (ret != 0) {
35          dev_err(i2c->dev, "cannot get bus (error %d)\n", ret);
36          s3c24xx_i2c_stop(i2c, -ENXIO);
37
38          iicstat = readl(i2c->regs + S3C2410_IICSTAT);
39
40          if ((iicstat & S3C2410_IICSTAT_BUSBUSY)){
41              iicstat &= ~(S3C2410_IICSTAT_TXRXEN | S3C2410_IICSTAT_BUSBUSY);
42              writel(iicstat, i2c->regs + S3C2410_IICSTAT);
43          }
44
45          msleep(1);
46
47          ret = -EAGAIN;
48          goto out;
49      }
50
51      spin_lock_irq(&i2c->lock);
52
53      i2c->msg      = msgs;
54      i2c->msg_num  = num;
55      i2c->msg_ptr  = 0;
56      i2c->msg_idx  = 0;
57      i2c->state    = STATE_START;
58
59      s3c24xx_i2c_enable_irq(i2c);

```



```
60 s3c24xx_i2c_message_start(i2c, msgs);
61 spin_unlock_irq(&i2c->lock);
62
63 timeout = wait_event_timeout(i2c->wait, i2c->msg_num == 0, Hz * 5);
64
65 ret = i2c->msg_idx;
66
67 if (timeout == 0)
68     dev_dbg(i2c->dev, "timeout\n");
69 else if (ret != num)
70     dev_dbg(i2c->dev, "incomplete xfer (%d)\n", ret);
71
72 msleep(1); /* 确保停止位已经被传递 */
73
74 out:
75 return ret;
76 }
77
78 static void s3c24xx_i2c_message_start(struct s3c24xx_i2c *i2c,
79                                     struct i2c_msg *msg)
80 {
81     unsigned int addr = (msg->addr & 0x7f) << 1;
82     unsigned long stat;
83     unsigned long iiccon;
84
85     stat = 0;
86     stat |= S3C2410_IICSTAT_TXRXEN;
87
88     if (msg->flags & I2C_M_RD) {
89         stat |= S3C2410_IICSTAT_MASTER_RX;
90         addr |= 1;
91     } else
92         stat |= S3C2410_IICSTAT_MASTER_TX;
93
94     if (msg->flags & I2C_M_REV_DIR_ADDR)
95         addr ^= 1;
96
97     s3c24xx_i2c_enable_ack(i2c);
98
99     iiccon = readl(i2c->regs + S3C2410_IICCON);
100     writel(stat, i2c->regs + S3C2410_IICSTAT);
101
102     dev_dbg(i2c->dev, "START: %08lx to IICSTAT, %02x to DS\n", stat, addr);
103     writeb(addr, i2c->regs + S3C2410_IICDS);
104
105     ndelay(i2c->tx_setup);
106
107     dev_dbg(i2c->dev, "iiccon, %08lx\n", iiccon);
108     writel(iiccon, i2c->regs + S3C2410_IICCON);
109
110     stat |= S3C2410_IICSTAT_START;
111     writel(stat, i2c->regs + S3C2410_IICSTAT);
112 }
```

s3c24xx_i2c_xfer()函数调用 s3c24xx_i2c_doxfer()函数传输 I²C 消息, 第 8 行的循环意味着最

多可以重试 `adap->retries` 次。

`s3c24xx_i2c_doxfer()` 首先将 S3C6410 的 I²C 适配器设置为 I²C 主设备, 其后初始化 `s3c24xx_i2c` 结构体, 使能 I²C 中断, 并调用 `s3c24xx_i2c_message_start()` 函数启动 I²C 消息的传输。

`s3c24xx_i2c_message_start()` 函数写 S3C6410 适配器对应的控制寄存器, 向 I²C 从设备传递开始位和从设备地址。

上述代码只是启动了 I²C 消息数组的传输周期, 并没有完整实现图 15.3 中给出的 `algorithm master_xfer` 流程。这个流程的完整实现需要借助 I²C 适配器上的中断来步步推进。代码清单 15.25 所示为 S3C6410 I²C 适配器中断处理函数以及其依赖的 `i2s_s3c_irq_nextbyte()` 函数的源代码。

代码清单 15.25 S3C6410 I²C 适配器中断处理函数

```

1  static irqreturn_t s3c24xx_i2c_irq(int irqno, void *dev_id)
2  {
3      struct s3c24xx_i2c *i2c = dev_id;
4      unsigned long status;
5      unsigned long tmp;
6
7      status = readl(i2c->regs + S3C2410_IICSTAT);
8      if (status & S3C2410_IICSTAT_ARBITR) {
9          ...
10     }
11
12     if (i2c->state == STATE_IDLE) {
13         tmp = readl(i2c->regs + S3C2410_IICCON);
14         tmp &= ~S3C2410_IICCON_IRQPEND;
15         writel(tmp, i2c->regs + S3C2410_IICCON);
16         goto out;
17     }
18
19     i2s_s3c_irq_nextbyte(i2c, status); /* 把传输工作进一步推进 */
20
21     out:
22         return IRQ_HANDLED;
23     }
24
25     static int i2s_s3c_irq_nextbyte(struct s3c24xx_i2c *i2c, unsigned long iicstat)
26     {
27         unsigned long tmp;
28         unsigned char byte;
29         int ret = 0;
30
31         switch (i2c->state) {
32             case STATE_IDLE:
33                 goto out;
34             break;
35         case STATE_STOP:
36             s3c24xx_i2c_disable_irq(i2c);
37             goto out_ack;
38         case STATE_START:
39             /* 我们最近做的一件事是启动一个新 I2C 消息 */
40             if (iicstat & S3C2410_IICSTAT_LASTBIT &&
41                 !(i2c->msg->flags & I2C_M_IGNORE_NAK)) {

```



```
42         /* 没有收到 ACK */
43         s3c24xx_i2c_stop(i2c, -EREMOTEIO);
44         goto out_ack;
45     }
46
47     if (i2c->msg->flags & I2C_M_RD)
48         i2c->state = STATE_READ;
49     else
50         i2c->state = STATE_WRITE;
51
52     /* 仅一条消息，而且长度为 0（主要用于适配器探测），发送停止位*/
53     if (is_lastmsg(i2c) && i2c->msg->len == 0) {
54         s3c24xx_i2c_stop(i2c, 0);
55         goto out_ack;
56     }
57
58     if (i2c->state == STATE_READ)
59         goto prepare_read;
60     /* 进入写状态 */
61     case STATE_WRITE:
62         ...
63     retry_write:
64         if (!is_msgend(i2c)) {
65             byte = i2c->msg->buf[i2c->msg_ptr++];
66             writeb(byte, i2c->regs + S3C2410_IICDS);
67             ndelay(i2c->tx_sep);
68         } else if (!is_lastmsg(i2c)) {
69             /* 推进到下一条消息 */
70             i2c->msg_ptr = 0;
71             i2c->msg_idx ++;
72             i2c->msg++;
73
74             /* 检查是否要为该消息产生开始位 */
75             if (i2c->msg->flags & I2C_M_NOSTART) {
76                 if (i2c->msg->flags & I2C_M_RD) {
77                     s3c24xx_i2c_stop(i2c, -EINVAL);
78                 }
79                 goto retry_write;
80             } else {
81                 /* 发送新的开始位 */
82                 s3c24xx_i2c_message_start(i2c, i2c->msg);
83                 i2c->state = STATE_START;
84             }
85         } else {
86             s3c24xx_i2c_stop(i2c, 0); /* send stop */
87         }
88     break;
89     case STATE_READ:
90     /* 有一个字节可读，看是否还有消息要处理 */
91     if (!(i2c->msg->flags & I2C_M_IGNORE_NAK) &&
92         !(is_msglast(i2c) && is_lastmsg(i2c))) {
93
94         if (iicstat & S3C2410_IICSTAT_LASTBIT) {
95             dev_dbg(i2c->dev, "READ: No Ack\n");
96         }
```



```

97         s3c24xx_i2c_stop(i2c, -ECONNREFUSED);
98         goto out_ack;
99     }
100 }
101 byte = readb(i2c->regs + S3C2410_IICDS);
102 i2c->msg->buf[i2c->msg_ptr++] = byte;
103
104 prepare_read:
105 if (is_msglast(i2c)) { /* last byte of buffer */
106     if (is_lastmsg(i2c))
107         s3c24xx_i2c_disable_ack(i2c);
108 } else if (is_msgend(i2c)) {
109     /* 还有消息要处理吗? */
110     if (is_lastmsg(i2c)) {
111         s3c24xx_i2c_stop(i2c, 0); /* last message, send stop and complete */
112     } else {
113         /* 推进到下一条消息 */
114         i2c->msg_ptr = 0;
115         i2c->msg_idx++;
116         i2c->msg++;
117     }
118 }
119 }
120 break;
121 }
122
123 /* irq 清除 */
124 out_ack:
125 tmp = readl(i2c->regs + S3C2410_IICCON);
126 tmp &= ~S3C2410_IICCON_IRQPEND;
127 writel(tmp, i2c->regs + S3C2410_IICCON);
128 out:
129 return ret;
130 }

```

中断处理函数 `s3c24xx_i2c_irq()` 主要通过调用 `i2s_s3c_irq_nextbyte()` 函数进行传输工作的进一步推进。`i2s_s3c_irq_nextbyte()` 函数通过 `switch(i2c->state)` 语句分成 `i2c->state` 的不同状态进行处理, 在每种状态下, 先检查 `i2c->state` 的状态与硬件寄存器应该处于的状态是否一致, 如果不一致, 则证明有误, 直接返回。当 I²C 处于读状态 `STATE_READ` 或写状态 `STATE_WRITE` 时, 通过 `is_lastmsg()` 函数判断是否传输的是最后一条 I²C 消息, 如果是, 则产生停止位, 否则通过 `i2c->msg_idx++`、`i2c->msg++` 推进到下一条消息。

15.6 AT24XX EEPROM 的 I²C 设备驱动实例

`drivers/i2c/chips/at24.c` 文件支持大多数 I²C 接口的 EEPROM, 正如我们之前所述, 一个具体的 I²C 设备驱动由两部分组成, 一部分是 `i2c_driver`, 用于将设备挂接于 I²C 总线, 一类是设备本身的驱动。对于 EEPROM 而言, 设备本身的驱动以 `bin_attribute` 二进制 `sysfs` 结点形式呈现。代码清单 15.26 给出了该驱动的框架。



代码清单 15.26 AT24XX EEPROM 驱动

```
1  /* bin_attribute 部分 */
2
3  static ssize_t at24_bin_read(struct kobject *kobj, struct bin_attribute *attr,
4      char *buf, loff_t off, size_t count)
5  {
6      struct at24_data *at24;
7      ssize_t retval = 0;
8
9      at24 = dev_get_drvdata(container_of(kobj, struct device, kobj));
10
11     ...
12
13     while (count) {
14         ssize_t status;
15
16         status = at24_eeprom_read(at24, buf, off, count);
17         ...
18     }
19
20     return retval;
21 }
22
23 static ssize_t at24_bin_write(struct kobject *kobj, struct bin_attribute *attr,
24     char *buf, loff_t off, size_t count)
25 {
26     struct at24_data *at24;
27     ssize_t retval = 0;
28
29     at24 = dev_get_drvdata(container_of(kobj, struct device, kobj));
30
31     ...
32
33     while (count) {
34         ssize_t status;
35
36         status = at24_eeprom_write(at24, buf, off, count);
37         ...
38     }
39
40     ...
41 }
42
43 /* i2c_driver 部分 */
44
45 static const struct i2c_device_id at24_ids[] = {
46     { "24c00", AT24_DEVICE_MAGIC(128 / 8, AT24_FLAG_TAKE8ADDR) },
47     { "24c01", AT24_DEVICE_MAGIC(1024 / 8, 0) },
48     ...
49     { "at24", 0 },
50     { /* END OF LIST */ }
51 };
52 MODULE_DEVICE_TABLE(i2c, at24_ids);
```

```

53
54 static int at24_probe(struct i2c_client *client, const struct i2c_device_id *id)
55 {
56 ...
57 /* 以 sysfs 二进制结点的形式呈现 eeprom 数据 */
58 at24->bin.attr.name = "eeprom";
59 at24->bin.attr.mode = chip.flags & AT24_FLAG_IRUGO ? S_IRUGO : S_IRUSR;
60 at24->bin.read = at24_bin_read;
61 at24->bin.size = chip.byte_len;
62 ...
63 at24->bin.write = at24_bin_write;
64
65 ...
66 err = sysfs_create_bin_file(&client->dev.kobj, &at24->bin);
67 if (err)
68     goto err_clients;
69
70 i2c_set_clientdata(client, at24);
71
72 ...
73 }
74
75 static int __devexit at24_remove(struct i2c_client *client)
76 {
77 struct at24_data *at24;
78 int i;
79
80 at24 = i2c_get_clientdata(client);
81 sysfs_remove_bin_file(&client->dev.kobj, &at24->bin);
82
83 for (i = 1; i < at24->num_addresses; i++)
84     i2c_unregister_device(at24->client[i]);
85
86 ...
87 }
88
89 static struct i2c_driver at24_driver = {
90 .driver = {
91     .name = "at24",
92     .owner = THIS_MODULE,
93 },
94 .probe = at24_probe,
95 .remove = __devexit_p(at24_remove),
96 .id_table = at24_ids,
97 };
98
99 static int __init at24_init(void)
100 {
101 io_limit = rounddown_pow_of_two(io_limit);
102 return i2c_add_driver(&at24_driver);
103 }
104 module_init(at24_init);
105
106 static void __exit at24_exit(void)
107 {

```



```
108 i2c_del_driver(&at24_driver);
109 }
110 module_exit(at24_exit);
```

上述代码中的 1~40 行对应 EEPROM 驱动本身的读写实现即 bin_attribute 驱动, 之后的一部分是 i2c_driver, 两者在 i2c_driver 的 probe()、remove() 函数中建立关联。i2c_driver 的 probe() 函数中初始化并通过第 66 行的 sysfs_create_bin_file() 注册了二进制 sysfs 结点, 而 remove() 函数则通过第 81 行的 sysfs_remove_bin_file() 注销了 sysfs 结点。

第 16 行调用的 at24_eeprom_read() 和第 36 行调用的 at24_eeprom_write() 通过 i2c_msg 和 i2c_transfer 完成实际的数据传输。

drivers/i2c/chips/at24.c 不依赖于具体的 CPU 和 I²C 控制器硬件特性, 因此, 如果某一电路板包含该外设, 只需要在板文件中添加对应的 i2c_board_info, 如对于 LDD6410 在 arch/arm/mach-s3c6410/mach-ldd6410.c 中添加的信息为:

```
static struct i2c_board_info i2c_devs0[] __initdata = {
    { I2C_BOARD_INFO("24c02", 0x50), },
};
```

此后, 我们在 LDD6410 上透过 /sys/class/i2c-adapter/i2c-0/0-0050/eeprom 文件结点即可操作连接的 EEPROM。

15.7 总结

Linux I²C 驱动体系结构有相当的复杂度, 它主要由 3 部分组成, 即 I²C 核心、I²C 总线驱动和 I²C 设备驱动。I²C 核心是 I²C 总线驱动和 I²C 设备驱动的中间枢纽, 它以通用的、与平台无关的接口实现了 I²C 中设备与适配器的沟通。I²C 总线驱动填充 i2c_adapter 和 i2c_algorithm 结构体, I²C 设备驱动填充 i2c_driver 结构体并实现其本身所对应设备类型的驱动。

另外, 系统中 i2c-dev.c 文件定义的主设备号为 89 的设备可以方便地给应用程序提供读写 I²C 设备寄存器的能力, 使得工程师大多数时候并不需要为具体的 I²C 设备驱动定义文件操作接口。

LINUX

第16章

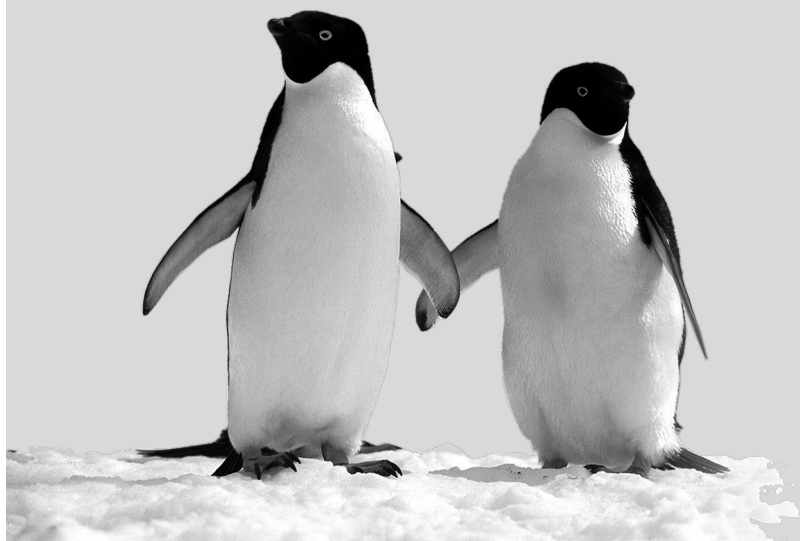
Linux 网络设备驱动

网络设备是完成用户数据包在网络媒介上发送和接收的设备，它将上层协议传递下来的数据包以特定的媒介访问控制方式进行发送，并将接收到的数据包传递给上层协议。

与字符设备和块设备不同，网络设备并不对应于/dev 目录下的文件，应用程序最终使用套接字（socket）完成与网络设备的接口。因而在网络设备身上并不能体现出“一切都是文件”的思想。

Linux 系统对网络设备驱动定义了4个层次，这4个层次为网络协议接口层、网络设备接口层、提供实际功能的设备驱动功能层和网络设备与媒介层。

在本章中，16.1 节讲解 Linux 网络设备驱动的层次结构，描述其4个层次各自的作用以及它们是如何协同合作以实现向下驱动网络设备硬件、向上提供数据包收发接口能力的。16.2~16.8 节主要讲解设备驱动功能层的各主要函数和数据结构，包括设备注册与注销、设备初始化、数据包收发函数、打开与释放函数等，在分析的基础上给出了抽象的设计模板。16.9 节介绍了DM9000 网卡的设备驱动及其在LDD6410 开发板上的移植。





16.1 Linux 网络设备驱动的结构

Linux 网络设备驱动程序的体系结构如图 16.1 所示, 从上到下可以划分为 4 层, 依次为网络协议接口层、网络设备接口层、提供实际功能的设备驱动功能层以及网络设备与媒介层, 这 4 层的作用如下所示。

(1) 网络协议接口层向网络层协议提供统一的数据包收发接口, 不论上层协议为 ARP 还是 IP, 都通过 `dev_queue_xmit()` 函数发送数据, 并通过 `netif_rx()` 函数接收数据。这一层的存在使得上层协议独立于具体的设备。

(2) 网络设备接口层向协议接口层提供统一的用于描述具体网络设备属性和操作的结构体 `net_device`, 该结构体是设备驱动功能层中各函数的容器。实际上, 网络设备接口层从宏观上规划了具体操作硬件的设备驱动功能层的结构。

(3) 设备驱动功能层各函数是网络设备接口层 `net_device` 数据结构的具体成员, 是驱使网络设备硬件完成相应动作的程序, 它通过 `hard_start_xmit()` 函数启动发送操作, 并通过网络设备上的中断触发接收操作。

(4) 网络设备与媒介层是完成数据包发送和接收的物理实体, 包括网络适配器和具体的传输媒介, 网络适配器被设备驱动功能层中的函数物理上驱动。对于 Linux 系统而言, 网络设备和媒介都可以是虚拟的。

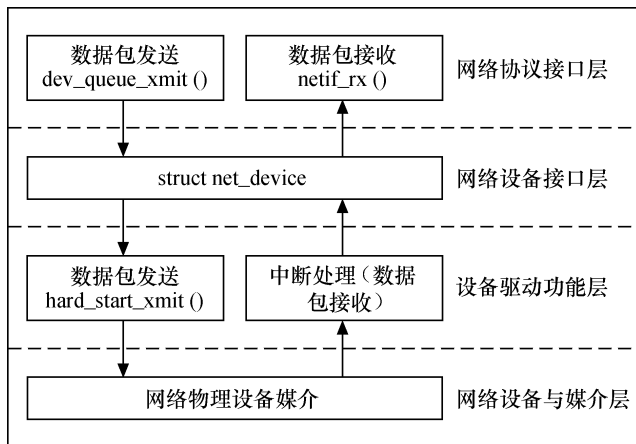


图 16.1 Linux 网络设备驱动程序的体系结构

在设计具体的网络设备驱动程序时, 我们需要完成的主要工作是编写设备驱动功能层的相关函数以填充 `net_device` 数据结构的内容并将 `net_device` 注册入内核。

16.1.1 网络协议接口层

网络协议接口层最主要的功能是给上层协议提供了透明的数据包发送和接收接口。当上层 ARP 或 IP 需要发送数据包时, 它将调用网络协议接口层的 `dev_queue_xmit()` 函数发送该数据

包，同时需传递给该函数一个指向 `struct sk_buff` 数据结构的指针。`dev_queue_xmit()`函数的原型为：

```
dev_queue_xmit (struct sk_buff * skb );
```

同样地，上层对数据包的接收也通过向 `netif_rx()`函数传递一个 `struct sk_buff` 数据结构的指针来完成。`netif_rx()`函数的原型为：

```
int netif_rx(struct sk_buff *skb);
```

`sk_buff` 结构体非常重要，定义于 `include/linux/skbuff.h` 文件，它的含义为“套接字缓冲区”，用于在 Linux 网络子系统各层之间传递数据，是 Linux 网络子系统数据传递的“中枢神经”。

当发送数据包时，Linux 内核的网络处理模块必须建立一个包含要传输的数据包的 `sk_buff`，然后将 `sk_buff` 递交给下层，各层在 `sk_buff` 中添加不同的协议头直至交给网络设备发送。同样地，当网络设备从网络媒介上接收到数据包后，它必须将接收到的数据转换为 `sk_buff` 数据结构并传递给上层，各层剥去相应的协议头直至交给用户。

代码清单 16.1 列出了 `sk_buff` 结构体的几个关键数据成员，其中 `head` 为整个缓冲区的头指针，`data` 为有效数据的头指针，`tail` 为其尾部位置，而 `transport_header`、`network_header`、`mac_header` 分别为传输层、网络层和 MAC 层的包头位置。

代码清单 16.1 套接字缓冲区 `sk_buff`

```
1 struct sk_buff {
2     ...
3     unsigned int      len,
4                       data_len;
5     __u16             mac_len,
6                       hdr_len;
7     ...
8     sk_buff_data_t    transport_header;
9     sk_buff_data_t    network_header;
10    sk_buff_data_t    mac_header;
11    ...
12    sk_buff_data_t    tail;
13    sk_buff_data_t    end;
14    unsigned char      *head,
15                      *data;
16 };
```

下面我们来分析套接字缓冲区涉及的操作函数，Linux 套接字缓冲区支持分配、释放、变更等功能函数。

(1) 分配。

Linux 内核用于分配套接字缓冲区的函数有：

```
struct sk_buff *alloc_skb(unsigned int len, gfp_t priority)
struct sk_buff *dev_alloc_skb(unsigned int len);
```

`alloc_skb()`函数分配一个套接字缓冲区和一个数据缓冲区，参数 `len` 为数据缓冲区的空间大小，通常以 `L1_CACHE_BYTES` 字节（对于 ARM 为 32）对齐，参数 `priority` 为内存分配的优先级。`dev_alloc_skb()`函数以 `GFP_ATOMIC` 优先级进行 `skb` 的分配。

(2) 释放。

Linux 内核用于释放套接字缓冲区的函数有：

```
void kfree_skb(struct sk_buff *skb);
void dev_kfree_skb(struct sk_buff *skb);
```



```
void dev_kfree_skb_irq(struct sk_buff *skb);
void dev_kfree_skb_any(struct sk_buff *skb);
```

上述函数用于释放被 `alloc_skb()` 函数分配的套接字缓冲区和数据缓冲区。

Linux 内核内部使用 `kfree_skb()` 函数, 而网络设备驱动程序中则最好用 `dev_kfree_skb()`、`dev_kfree_skb_irq()` 或 `dev_kfree_skb_any()` 函数进行套接字缓冲区的释放。其中, `dev_kfree_skb()` 函数用于非中断上下文, `dev_kfree_skb_irq()` 函数用于中断上下文, 而 `dev_kfree_skb_any()` 函数则在中断和非中断上下文中皆可采用。

(3) 变更。

Linux 内核中可以用如下函数在缓冲区尾部增加数据:

```
unsigned char *skb_put(struct sk_buff *skb, unsigned int len);
```

它会导致 `skb->tail` 后移 `len`, 而 `skb->len` 增加 `len` 的大小。通常, 设备驱动的收数据处理中会调用此函数。

Linux 内核中可以用如下函数在缓冲区开头增加数据:

```
unsigned char *skb_push(struct sk_buff *skb, unsigned int len);
```

它会导致 `skb->data` 前移 `len`, 而 `skb->len` 增加 `len` 的大小。与该函数完成相反功能的函数是 `skb_pull()`, 它可以在缓冲区开头移除数据。

对于一个空的缓冲区而言, 调用如下函数可以调整缓冲区的头部:

```
static inline void skb_reserve(struct sk_buff *skb, int len);
```

它会将 `skb->data` 和 `skb->tail` 同时后移 `len`。

16.1.2 网络设备接口层

网络设备接口层的主要功能是为千变万化的网络设备定义了统一、抽象的数据结构 `net_device` 结构体, 以不变应万变, 实现多种硬件在软件层次上的统一。

`net_device` 结构体在内核中指代一个网络设备, 定义于 `include/linux/netdevice.h` 文件, 网络设备驱动程序只需通过填充 `net_device` 的具体成员并注册 `net_device` 即可实现硬件操作函数与内核的挂接。

`net_device` 是一个巨大的结构体, 包含网络设备的属性描述和操作接口, 下面介绍其中的一些关键成员。

(1) 全局信息。

```
char name[IFNAMESIZ];
```

`name` 是网络设备的名称。

```
int (*init)(struct net_device *dev);
```

`init` 为设备初始化函数指针, 如果这个指针被设置了, 则网络设备被注册时将调用该函数完成对 `net_device` 结构体的初始化。但是, 设备驱动程序可以不实现这个函数并将其赋值为 `NULL`。

(2) 硬件信息。

```
unsigned long mem_end;
unsigned long mem_start;
```

`mem_start` 和 `mem_end` 分别定义了设备所使用的共享内存的起始和结束地址。

```
unsigned long base_addr;
unsigned char irq;
unsigned char if_port;
unsigned char dma;
```


`base_addr` 为网络设备 I/O 基地址。

`irq` 为设备使用的中断号。

`if_port` 指定多端口设备使用哪一个端口，该字段仅针对多端口设备。例如，如果设备同时支持 `IF_PORT_10BASE2`（同轴电缆）和 `IF_PORT_10BASET`（双绞线），则可使用该字段。

`dma` 指定分配给设备的 DMA 通道。

(3) 接口信息。

```
unsigned short hard_header_len;
```

`hard_header_len` 是网络设备的硬件头长度，在以太网设备的初始化函数中，该成员被赋为 `ETH_HLEN`，即 14。

```
unsigned short type;
```

`type` 是接口的硬件类型。

```
unsigned mtu;
```

`mtu` 指最大传输单元（MTU）。

```
unsigned char dev_addr[MAX_ADDR_LEN];
```

```
unsigned char broadcast[MAX_ADDR_LEN];
```

`dev_addr[]`、`broadcast[]` 无符号字符数组，分别用于存放设备的硬件地址和广播地址。对于以太网而言，这两个地址的长度都为 6 个字节。以太网设备的广播地址为 6 个 `0xFF`，而 MAC 地址需由驱动程序从硬件上读出并填充到 `dev_addr[]` 中。

```
unsigned short flags;
```

`flags` 指网络接口标志，以 `IFF_`（interface flags）开头，部分标志由内核来管理，其他的在接口初始化时被设置以说明设备接口的能力和特性。接口标志包括 `IFF_UP`（当设备被激活并可以开始发送数据包时，内核设置该标志）、`IFF_AUTOMEDIA`（设备可在多种媒介间切换）、`IFF_BROADCAST`（允许广播）、`IFF_DEBUG`（调试模式，可用于控制 `printk` 调用的详细程度）、`IFF_LOOPBACK`（回环）、`IFF_MULTICAST`（允许组播）、`IFF_NOARP`（接口不能执行 ARP）、`IFF_POINTOPOINT`（接口连接到点到点链路）等。

(4) 设备操作函数。

```
int (*open)(struct net_device *dev);
```

```
int (*stop)(struct net_device *dev);
```

`open()` 函数的作用是打开网络接口设备，获得设备需要的 I/O 地址、IRQ、DMA 通道等。`stop()` 函数的作用是停止网络接口设备，与 `open()` 函数的作用相反。

```
int (*hard_start_xmit)(struct sk_buff *skb, struct net_device *dev);
```

`hard_start_xmit()` 函数会启动数据包的发送，当系统调用驱动程序的 `hard_start_xmit()` 函数时，需要向其传入一个 `sk_buff` 结构体指针，以使得驱动程序能获取从上层传递下来的数据包。

```
void (*tx_timeout)(struct net_device *dev);
```

当数据包的发送超时，`tx_timeout()` 函数会被调用，该函数需采取重新启动数据包发送过程或重新启动硬件等措施来恢复网络设备到正常状态。

```
int (*hard_header)(struct sk_buff *skb,
                    struct net_device *dev,
                    unsigned short type,
                    void *daddr,
                    void *saddr,
                    unsigned len);
```

`hard_header()` 函数完成硬件帧头填充，返回填充的字节数。传入该函数的参数包括 `sk_buff` 指



针、设备指针、协议类型、目的地址、源地址以及数据长度。对于以太网设备而言,将内核提供的 `eth_header()` 函数赋值给 `hard_header` 指针即可。

```
struct net_device_stats* (*get_stats)(struct net_device *dev);
```

`get_stats()` 函数用于获得网络设备的状态信息,它返回一个 `net_device_stats` 结构体。`net_device_stats` 结构体保存了网络设备详细的流量统计信息,如发送和接收到的数据包数、字节数等,详见 16.8 节。

```
int (*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);
int (*set_config)(struct net_device *dev, struct ifmap *map);
int (*set_mac_address)(struct net_device *dev, void *addr);
```

`do_ioctl()` 函数用于进行设备特定的 I/O 控制。

`set_config()` 函数用于配置接口,可用于改变设备的 I/O 地址和中断号。

`set_mac_address()` 函数用于设置设备的 MAC 地址。

```
const struct ethtool_ops *ethtool_ops;
```

上述结构体中的一系列成员函数用于更改或报告网络设备的设置,主要包括 `get_settings`、`set_settings`、`get_wol`、`set_wol`、`get_eeprom`、`get_ringparam`、`set_ringparam` 等成员函数。

```
void (*poll_controller)(struct net_device *dev);
```

在内核配置了 `NET_POLL_CONTROLLER` 的情况下,为了支持纯粹的 `netconsole`(如用于 `kgdb` 调试),可以为网络设备驱动实现 `poll_controller()` 成员函数,它完全以轮询方式接收数据包。

`net_device` 结构体的上述成员需要根据对应网络设备的具体情况在设备初始化时被填充,详见 16.3 节。

(5) 辅助成员。

```
unsigned long trans_start;
unsigned long last_rx;
```

`trans_start` 记录最后的数据包开始发送时的时间戳, `last_rx` 记录最后一次接收到数据包时的时间戳,这两个时间戳记录的都是 `jiffies`,驱动程序应维护这两个成员。

```
void *priv;
```

`priv` 为设备的私有信息指针,与 `filp->private_data` 的地位相当。设备驱动程序中应该以 `netdev_priv()` 函数获得该指针。

通常情况下,网络设备驱动以中断方式接收数据包,而 `poll_controller()` 则采用纯轮询方式,另外一种数据接收方式是 NAPI (New API) 其数据接收流程如下为“接收中断来临——关闭接收中断——以轮询方式接收所有数据包直到收空——开启接收中断——接收中断来临……”。内核中提供了如下与 NAPI 相关的 API:

```
static inline void netif_napi_add(struct net_device *dev,
                                struct napi_struct *napi,
                                int (*poll)(struct napi_struct *, int),
                                int weight);
static inline void netif_napi_del(struct napi_struct *napi);
```

以上两个函数分别用于初始化和移除一个 NAPI, `netif_napi_add()` 的 `poll` 参数是 NAPI 要调度执行的轮询函数。

```
static inline void napi_enable(struct napi_struct *n);
static inline void napi_disable(struct napi_struct *n);
```

以上两个函数分别用于使能和禁止 NAPI 调度。

```
static inline int napi_schedule_prep(struct napi_struct *n);
```

该函数用于检查 NAPI 是否可以调度，而 `napi_schedule()` 函数用于调度轮询实例的运行，其原型为：

```
static inline void napi_schedule(struct napi_struct *n);
```

在 NAPI 处理完成的时候应该调用：

```
static inline void napi_complete(struct napi_struct *n);
```

16.1.3 设备驱动功能层

`net_device` 结构体的成员（属性和函数指针）需要被设备驱动功能层的具体数值和函数赋予。对于具体的设备 `xxx`，工程师应该编写设备驱动功能层的函数，这些函数形如 `xxx_open()`、`xxx_stop()`、`xxx_tx()`、`xxx_hard_header()`、`xxx_get_stats()`、`xxx_tx_timeout()` 等。

由于网络数据包的接收可由中断引发，设备驱动功能层中另一个主体部分将是中断处理函数，它负责读取硬件上接收的数据包并传送给上层协议，可能包含 `xxx_interrupt()` 和 `xxx_rx()` 函数，前者完成中断类型判断等基本的工作，后者则需完成数据包的生成和递交上层等复杂工作。

16.2~16.8 节将对上述函数进行详细分析并给出参考设计模板。

对于特定的设备，我们还可以定义其相关私有数据和操作，并封装为一个私有信息结构体 `xxx_private`，让其指针被赋值给 `net_device` 的 `priv` 成员。`xxx_private` 结构体中可包含设备特殊的属性和操作、自旋锁与信号量、定时器以及统计信息等，由工程师自定义。

16.1.4 网络设备与媒介层

网络设备与媒介层直接对应于实际的硬件设备。为了给设备的物理配置和寄存器操作一个更一般的描述，我们可以定义一组宏和一组访问设备内部寄存器的函数，具体的宏和函数与特定的硬件紧密相关。代码清单 16.2 所示为相应的设计范例。

代码清单 16.2 网络设备底层硬件操作

```
1  /* 寄存器定义 */
2  #define DATA_REG 0x0004
3  #define CMD_REG 0x0008
4
5  /* 寄存器读写函数 */
6  static u16 xxx_readword(u32 base_addr, int portno)
7  {
8      ... /*读取寄存器的值并返回*/
9  }
10
11 static void xxx_writeword(u32 base_addr, int portno, u16 value)
12 {
13     ... /*向寄存器写入数值*/
14 }
```

16.2 网络设备驱动的注册与注销

网络设备驱动的注册与注销使用成对出现的 `register_netdev()` 和 `unregister_netdev()` 函数完成，



这两个函数的原型为:

```
int register_netdev(struct net_device *dev);
void unregister_netdev(struct net_device *dev);
```

这两个函数都接收一个 `net_device` 结构体指针为参数, 可见 `net_device` 数据结构在网络设备驱动中的核心地位。

`net_device` 的生成和成员的赋值并非一定要由工程师逐个亲自动手完成, 可以利用下面的宏帮助我们填充:

```
#define alloc_netdev(sizeof_priv, name, setup) \
    alloc_netdev_mq(sizeof_priv, name, setup, 1)
#define alloc_etherdev(sizeof_priv) alloc_etherdev_mq(sizeof_priv, 1)
```

`alloc_netdev` 宏引用的 `alloc_etherdev_mq()` 函数的原型为:

```
struct net_device *alloc_netdev_mq(int sizeof_priv, const char *name,
    void (*setup)(struct net_device *), unsigned int queue_count);
```

`alloc_netdev_mq()` 函数生成一个 `net_device` 结构体, 对其成员赋值并返回该结构体的指针。第一个参数为设备私有成员的大小, 第二个参数为设备名, 第三个参数为 `net_device` 的 `setup()` 函数指针, 第四个参数为要分配的子队列的数量。`setup()` 函数接收的参数也为 `struct net_device` 指针, 用于预置 `net_device` 成员的值。

`alloc_etherdev()` 是 `alloc_netdev()` 针对以太网的“快捷”函数, 这从 `alloc_etherdev()` 引用的 `alloc_etherdev_mq()` 函数的源代码可以看出, 如代码清单 16.3 所示。

代码清单 16.3 `alloc_etherdev()` 函数

```
1 struct net_device *alloc_etherdev_mq(int sizeof_priv, unsigned int queue_count)
2 {
3     /* 以 ether_setup 为 alloc_netdev_mq 的 setup 参数 */
4     return alloc_netdev_mq(sizeof_priv, "eth%d", ether_setup, queue_count);
5 }
```

上述代码中的第 4 行传入 `alloc_netdev_mq()` 函数的第三个参数为 `ether_setup()` 函数地址, `ether_setup()` 是由 Linux 内核提供的一个对以太网设备 `net_device` 结构体中公有成员快速赋值的函数。

完成与 `alloc_enetdev()` 和 `alloc_etherdev()` 函数相反功能, 即释放 `net_device` 结构体的函数为:

```
void free_netdev(struct net_device *dev);
```

`net_device` 结构体的分配和网络设备驱动注册需在网络设备驱动程序的模块加载函数中进行, 而 `net_device` 结构体的释放和网络设备驱动的注销则需在模块卸载函数中完成, 如代码清单 16.4 所示。

代码清单 16.4 网络设备驱动程序的模块加载函数模板

```
1 int xxx_init_module(void)
2 {
3     ...
4     /* 分配 net_device 结构体并对其成员赋值 */
5     xxx_dev = alloc_netdev(sizeof(struct xxx_priv), "sn%d", xxx_init);
6     if (xxx_dev == NULL)
7         ... /* 分配 net_device 失败 */
8
9     /* 注册 net_device 结构体 */
10    if ((result = register_netdev(xxx_dev)))
```

```

11     ...
12 }
13
14 void xxx_cleanup(void)
15 {
16     ...
17     /* 注销 net_device 结构体 */
18     unregister_netdev(xxx_dev);
19     /* 释放 net_device 结构体 */
20     free_netdev(xxx_dev);
21 }

```

16.3 网络设备的初始化

网络设备的初始化主要需要完成如下几个方面的工作。

- 进行硬件上的准备工作，检查网络设备是否存在，如果存在，则检测设备所使用的硬件资源。
- 进行软件接口上的准备工作，分配 `net_device` 结构体并对其数据和函数指针成员赋值。
- 获得设备的私有信息指针并初始化其各成员的值。如果私有信息中包括自旋锁或信号量等并发或同步机制，则需对其进行初始化。

对 `net_device` 结构体成员及私有数据的赋值都可能需要与硬件初始化工作协同进行，即硬件检测出了相应的资源，需要根据检测结果填充 `net_device` 结构体成员和私有数据。

一个网络设备驱动初始化函数的模板如代码清单 16.5 所示，具体的设备驱动初始化函数并不一定完全和本模板一样，但是其本质过程是一致的。

代码清单 16.5 网络设备驱动的初始化函数模板

```

1 void xxx_init(struct net_device *dev)
2 {
3     /*设备的私有信息结构体*/
4     struct xxx_priv *priv;
5
6     /* 检查设备是否存在和设备所使用的硬件资源 */
7     xxx_hw_init();
8
9     /* 初始化以太网设备的公用成员 */
10    ether_setup(dev);
11
12    /*设置设备的成员函数指针*/
13    dev->open = xxx_open;
14    dev->stop = xxx_release;
15    dev->set_config = xxx_config;
16    dev->hard_start_xmit = xxx_tx;
17    dev->do_ioctl = xxx_ioctl;
18    dev->get_stats = xxx_stats;
19    dev->change_mtu = xxx_change_mtu;
20    dev->rebuild_header = xxx_rebuild_header;
21    dev->hard_header = xxx_header;

```



```
22 dev->tx_timeout = xxx_tx_timeout;
23 dev->watchdog_timeo = timeout;
24
25 /* 取得私有信息, 并初始化它*/
26 priv = netdev_priv(dev);
27 ... /* 初始化设备私有数据区 */
28 }
```

上述代码第 7 行的 `xxx_hw_init()` 函数完成硬件相关的初始化操作如下。

- 探测 xxx 网络设备是否存在。探测的方法类似于数学上的“反证法”，即先假设存在设备 xxx，访问该设备，如果设备的表现与预期的一致，就确定设备存在；否则，假设错误，设备 xxx 不存在。
- 探测设备的具体硬件配置。一些设备驱动编写得非常通用，对于同类的设备使用统一的驱动，我们需要在初始化时探测设备的具体型号。另外，即便是同一设备，在硬件上的配置也可能不一样，我们也可以探测设备所使用的硬件资源。
- 申请设备所需要的硬件资源，如用 `request_region()` 函数进行 I/O 端口的申请等，但是这个过程可以放在设备的打开函数 `xxx_open()` 中完成。

16.4 网络设备的打开与释放

网络设备的打开函数需要完成如下工作。

- 使能设备使用的硬件资源，申请 I/O 区域、中断和 DMA 通道等。
- 调用 Linux 内核提供的 `netif_start_queue()` 函数，激活设备发送队列。

网络设备的关闭函数需要完成如下工作。

- 调用 Linux 内核提供的 `netif_stop_queue()` 函数，停止设备传输包。
- 释放设备所使用的 I/O 区域、中断和 DMA 资源。

Linux 内核提供的 `netif_start_queue()` 和 `netif_stop_queue()` 两个函数的原型为：

```
void netif_start_queue(struct net_device *dev);
void netif_stop_queue (struct net_device *dev);
```

根据以上分析，可得出如代码清单 16.6 所示的网络设备打开和释放函数的模板。

代码清单 16.6 网络设备打开和释放函数模板

```
1 int xxx_open(struct net_device *dev)
2 {
3     /* 申请端口、IRQ 等，类似于 fops->open */
4     ret = request_irq(dev->irq, &xxx_interrupt, 0, dev->name, dev);
5     ...
6     netif_start_queue(dev);
7     ...
8 }
9
10 int xxx_release(struct net_device *dev)
11 {
12     /* 释放端口、IRQ 等，类似于 fops->close */
13     free_irq(dev->irq, dev);
```

```

14     ...
15     netif_stop_queue(dev); /* can't transmit any more */
16     ...
17 }

```

16.5 数据发送流程

从 16.1 节网络设备驱动程序的结构分析可知，Linux 网络子系统在发送数据包时，会调用驱动程序提供的 `hard_start_transmit()` 函数，该函数用于启动数据包的发送。在设备初始化的时候，这个函数指针需被初始化指向设备的 `xxx_tx()` 函数。

网络设备驱动完成数据包发送的流程如下。

- (1) 网络设备驱动程序从上层协议传递过来的 `sk_buff` 参数获得数据包的有效数据和长度，将有效数据放入临时缓冲区。
 - (2) 对于以太网，如果有效数据的长度小于以太网冲突检测所要求数据帧的最小长度 `ETH_ZLEN`，则给临时缓冲区的末尾填充 0。
 - (3) 设置硬件的寄存器，驱使网络设备进行数据发送操作。
- 完成以上 3 个步骤的网络设备驱动程序的数据包发送函数的模板如代码清单 16.7 所示。

代码清单 16.7 网络设备驱动程序的数据包发送函数模板

```

1  int xxx_tx(struct sk_buff *skb, struct net_device *dev)
2  {
3      int len;
4      char *data, shortpkt[ETH_ZLEN];
5      if (xxx_send_available(...)) { /* 发送队列未满，可以发送 */
6          /* 获得有效数据指针和长度 */
7          data = skb->data;
8          len = skb->len;
9          if (len < ETH_ZLEN) {
10             /* 如果帧长小于以太网帧最小长度，补 0 */
11             memset(shortpkt, 0, ETH_ZLEN);
12             memcpy(shortpkt, skb->data, skb->len);
13             len = ETH_ZLEN;
14             data = shortpkt;
15         }
16     }
17     dev->trans_start = jiffies; /* 记录发送时间戳 */
18
19     /* 设置硬件寄存器让硬件把数据包发送出去 */
20     xxx_hw_tx(data, len, dev);
21     ...
22 } else {
23     netif_stop_queue(dev);
24     ...
25 }
26 }

```

这里特别要强调第 23 行对 `netif_stop_queue()` 的调用，当发送队列为满或因其他原因来不及发送当前上层传下来的包，则调用此函数阻止上层继续向网络设备驱动传递数据包。当忙于发送的



数据包被发送完成后, TX 结束的中断处理中, 应该调用 `netif_wake_queue()` 唤醒被阻塞的上层, 以启动它继续向网络设备驱动传送数据包。

当数据传输超时时, 意味着当前的发送操作失败或硬件已陷入未知状态, 此时, 数据包发送超时处理函数 `xxx_tx_timeout()` 将被调用。这个函数也需要调用 Linux 内核提供的 `netif_wake_queue()` 函数重新启动设备发送队列, 如代码清单 16.8 所示。

代码清单 16.8 网络设备驱动程序的数据包发送超时函数模板

```
1 void xxx_tx_timeout(struct net_device *dev)
2 {
3     ...
4     netif_wake_queue(dev); /* 重新启动设备发送队列 */
5 }
```

从前文可知, `netif_wake_queue()` 和 `netif_stop_queue()` 是数据发送流程中要调用的两个非常重要的函数, 分别用于唤醒和阻止上层向下传送数据包, 它们的原型定义于 `include/linux/netdevice.h`, 如下:

```
static inline void netif_wake_queue(struct net_device *dev);
static inline void netif_stop_queue(struct net_device *dev);
```

16.6 数据接收流程

网络设备接收数据的主要方法是由中断引发设备的中断处理函数, 中断处理函数判断中断类型, 如果为接收中断, 则读取接收到的数据, 分配 `sk_buffer` 数据结构和数据缓冲区, 将接收到的数据复制到数据缓冲区, 并调用 `netif_rx()` 函数将 `sk_buffer` 传递给上层协议。代码清单 16.9 所示为完成这一过程的函数模板。

代码清单 16.9 网络设备驱动的中断处理函数模板

```
1 static void xxx_interrupt(int irq, void *dev_id)
2 {
3     ...
4     switch (status & ISQ_EVENT_MASK) {
5     case ISQ_RECEIVER_EVENT:
6         /* 获取数据包 */
7         xxx_rx(dev);
8         break;
9         /* 其他类型的中断 */
10    }
11 }
12 static void xxx_rx(struct xxx_device *dev)
13 {
14     ...
15     length = get_rev_len (...);
16     /* 分配新的套接字缓冲区 */
17     skb = dev_alloc_skb(length + 2);
18
19     skb_reserve(skb, 2); /* 对齐 */
20     skb->dev = dev;
```



```

21
22  /* 读取硬件上接收到的数据 */
23  insw(ioaddr + RX_FRAME_PORT, skb_put(skb, length), length >> 1);
24  if (length &1)
25      skb->data[length - 1] = inw(ioaddr + RX_FRAME_PORT);
26
27  /* 获取上层协议类型 */
28  skb->protocol = eth_type_trans(skb, dev);
29
30  /* 把数据包交给上层 */
31  netif_rx(skb);
32
33  /* 记录接收时间戳 */
34  dev->last_rx = jiffies;
35  ...
36  }

```

从上述代码的第 4~7 行可以看出，当设备的中断处理程序判断中断类型为数据包接收中断时，它调用第 12~36 定义的 `xxx_rx()` 函数完成更深入的数据包接收工作。`xxx_rx()` 函数代码中的第 15 行从硬件读取到接收数据包有效数据的长度，第 16~19 行分配 `sk_buff` 和数据缓冲区，第 22~25 行读取硬件上接收到的数据并放入数据缓冲区，第 27~28 行解析接收数据包上层协议的类型，最后，第 30~31 行代码将数据包上交给上层协议。

如果是 NAPI 兼容的设备驱动，则可以通过 `poll` 方式接收数据包。这种情况下，我们需要为该设备驱动提供作为 `netif_napi_add()` 参数的 `xxx_poll()` 函数，如代码清单 16.10 所示。

代码清单 16.10 网络设备驱动的 `poll` 函数模板

```

1  static int xxx_poll(struct napi_struct *napi, int budget)
2  {
3      int npackets = 0;
4      struct sk_buff *skb;
5      struct xxx_priv *priv = container_of(napi, struct xxx_priv, napi);
6      struct xxx_packet *pkt;
7
8      while (npackets < budget && priv->rx_queue) {
9          /*从队列中取出数据包*/
10         pkt = xxx_dequeue_buf(dev);
11
12         /*接下来的处理，和中断触发的数据包接收一致*/
13         skb = dev_alloc_skb(pkt->datalen + 2);
14         ...
15         skb_reserve(skb, 2);
16         memcpy(skb_put(skb, pkt->datalen), pkt->data, pkt->datalen);
17         skb->dev = dev;
18         skb->protocol = eth_type_trans(skb, dev);
19         /*调用 netif_receive_skb，而不是 net_rx，将数据包交给上层协议*/
20         netif_receive_skb(skb);
21
22         /*更改统计数据 */
23         priv->stats.rx_packets++;
24         priv->stats.rx_bytes += pkt->datalen;
25         xxx_release_buffer(pkt);
26         npackets++;

```



```
27     }
28     if (npackets < budget) {
29         napi_complete(napi);
30         xxx_enable_rx_int (...); /* 再次使能网络设备的接收中断 */
31     }
32     return npackets;
33 }
```

上述代码中的 `budget` 是在初始化阶段分配给接口的 `weight` 值, `poll` 函数每次只能接收最多 `budget` 数量的数据包。第 8 行的 `while()` 循环读取设备的接收缓冲区, 读取数据包并提交给上层。这个过程和中断触发的数据包接收过程一致, 但是最后使用 `netif_receive_skb()` 函数而非 `netif_rx()` 函数将数据包提交给上层。这里体现出了中断处理机制和轮询机制之间的差别。

当一个轮询过程结束时, 第 29 行代码调用 `napi_complete()` 宣布这一消息, 而第 30 行代码则再次启动网络设备的接收中断。

虽然 NAPI 兼容的设备驱动以 `poll` 方式接收数据包, 但是仍然需要首次数据包接收中断来触发 `poll` 过程。与数据包的中断接收方式不同的是, 以轮询方式接收数据包时, 当第一次中断发生后, 中断处理程序要禁止设备的数据包接收中断并调度 NAPI, 如代码清单 16.11 所示。

代码清单 16.11 网络设备驱动的 `poll` 中断处理函数模板

```
1 static void xxx_poll_interrupt(int irq, void *dev_id)
2 {
3     switch (status & ISQ_EVENT_MASK) {
4     case ISQ_RECEIVER_EVENT:
5         ... /* 获取数据包 */
6         xxx_disable_rx_int(...); /* 禁止接收中断 */
7         napi_schedule(&priv->napi);
8         break;
9         ... /* 其他类型的中断 */
10    }
11 }
```

上述代码第 7 行的 `napi_schedule()` 函数被轮询方式驱动的中断程序调用, 将设备的 `poll` 方法添加到网络层的 `poll` 处理队列中, 排队并且准备接收数据包, 最终触发一个 `NET_RX_SOFTIRQ` 软中断, 通知网络层接收数据包。图 16.2 所示为 NAPI 驱动程序各部分的调用关系。

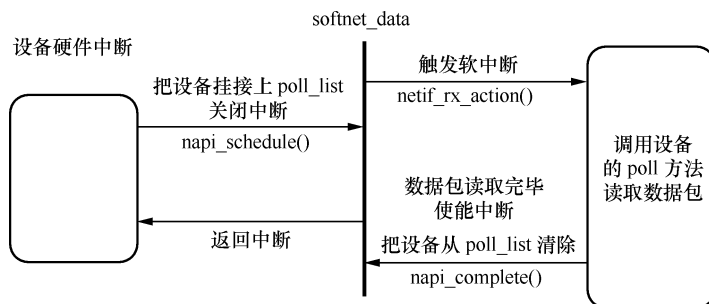


图 16.2 NAPI 调用关系

在支持 NAPI 的网络设备驱动中, 通常还会进行如下与 NAPI 相关的工作。

(1) 在私有数据结构体如 `xxx_priv` 中增加一个成员：

```
struct napi_struct napi;
```

这是代码清单 16.11 第 5 行调用 `container_of()` 可以得到 `xxx_priv` 类型指针的原因。

(2) 通常会在设备驱动初始化时调用：

```
netif_napi_add(dev, napi, xxx_poll, XXX_NET_NAPI_WEIGHT);
```

(3) 通常会在 `net_device` 结构体的 `open()` 和 `stop()` 成员函数中分别调用 `napi_enable()` 和 `napi_disable()`。

16.7 网络连接状态

网络适配器硬件电路可以检测出链路上是否有载波，载波反映了网络的连接是否正常。网络设备驱动可以通过 `netif_carrier_on()` 和 `netif_carrier_off()` 函数改变设备的连接状态，如果驱动检测到连接状态发生变化，也应该以 `netif_carrier_on()` 和 `netif_carrier_off()` 函数显式地通知内核。

除了 `netif_carrier_on()` 和 `netif_carrier_off()` 函数以外，另一个函数 `netif_carrier_ok()` 可用于向调用者返回链路路上的载波信号是否存在。

这几个函数都接收一个 `net_device` 设备结构体指针为参数，原型分别为：

```
void netif_carrier_on(struct net_device *dev);
void netif_carrier_off(struct net_device *dev);
int netif_carrier_ok(struct net_device *dev);
```

网络设备驱动程序中可采取一定的手段来检测和报告链路状态，例如设置一个定时器来对链路状态进行周期性地检查。当定时器到期之后，在定时器处理函数中读取物理设备的相关寄存器获得载波状态，从而更新设备的连接状态，如代码清单 16.12 所示。

代码清单 16.12 网络设备驱动用定时器周期检查链路状态

```
1 static void xxx_timer(unsigned long data)
2 {
3     struct net_device *dev = (struct net_device*)data;
4     u16 link;
5     ...
6     if (!(dev->flags & IFF_UP))
7         goto set_timer;
8
9     /* 获得物理上的连接状态 */
10    if (link = xxx_chk_link(dev)) {
11        if (!(dev->flags & IFF_RUNNING)) {
12            netif_carrier_on(dev);
13            dev->flags |= IFF_RUNNING;
14            printk(KERN_DEBUG "%s: link up\n", dev->name);
15        }
16    } else {
17        if (dev->flags & IFF_RUNNING) {
18            netif_carrier_off(dev);
19            dev->flags &= ~IFF_RUNNING;
20            printk(KERN_DEBUG "%s: link down\n", dev->name);
21        }
22    }
```



```
22     }
23
24     set_timer:
25     priv->timer.expires = jiffies + 1 * Hz;
26     priv->timer.data = (unsigned long)dev;
27     priv->timer.function = &xxx_timer; /* timer handler */
28     add_timer(&priv->timer);
29 }
```

上述代码第 10 行调用的 `xxx_chk_link()` 函数用于读取网络适配器硬件的相关寄存器以获得链路连接状态, 具体实现由硬件决定。当链路连接上时, 第 12 行的 `netif_carrier_on()` 函数显式地通知内核链路正常; 反之, 第 18 行的 `netif_carrier_off()` 同样显式地通知内核链路失去连接。

此外, 从上述源代码还可以看出, 定时器处理函数会不停地利用第 24~28 行代码启动新的定时器以实现周期检测的目的。那么最初启动定时器的地方在哪里呢? 很显然, 它最适合在设备的打开函数中完成, 如代码清单 16.13 所示。

代码清单 16.13 在网络设备驱动的打开函数中初始化定时器

```
1  static int xxx_open(struct net_device *dev)
2  {
3      struct xxx_priv *priv = (struct xxx_priv*)dev->priv;
4
5      ...
6      priv->timer.expires = jiffies + 3 * Hz;
7      priv->timer.data = (unsigned long)dev;
8      priv->timer.function = &xxx_timer; /* timer handler */
9      add_timer(&priv->timer);
10     ...
11 }
```

16.8 参数设置和统计数据

在网络设备的驱动程序中还提供一些方法供系统对设备的参数进行设置或读取设备相关的信息。

当用户调用 `ioctl()` 函数, 并指定 `SIOCSIFHWADDR` 命令时, 意味着要设置这个设备的 MAC 地址。设置网络设备的 MAC 地址可用如代码清单 16.14 所示的模板。

代码清单 16.14 设置网络设备的 MAC 地址

```
1  static int set_mac_address(struct net_device *dev, void *addr)
2  {
3      if (netif_running(dev))
4          return -EBUSY; /* 设备忙 */
5
6      /* 设置以太网的 MAC 地址 */
7      xxx_set_mac(dev, addr);
8
9      return 0;
10 }
```

上述程序首先用 `netif_running()` 宏判断设备是否正在运行, 如果是, 则意味着设备忙, 此时

不允许设置 MAC 地址；否则，调用 `xxx_set_mac()` 函数在网络适配器硬件内写入新的 MAC 地址。这要求设备在硬件上支持 MAC 地址的修改，而实际上，许多设备并不提供修改 MAC 地址的接口。

`netif_running()` 宏的定义为：

```
#define netif_running(dev) (dev->flags & IFF_UP)
```

当用户调用 `ioctl()` 函数时，若命令为 `SIOCSIFMAP`（如在控制台中运行网络配置命令 `ifconfig` 就会引发这一调用），系统会调用驱动程序的 `set_config()` 函数。

系统会向 `set_config()` 函数传递一个 `ifmap` 结构体，该结构体中主要包含用户欲设置的设备要使用的 I/O 地址、中断等信息。注意，并非 `ifmap` 结构体中给出的所有修改都是可以接受的。实际上，大多数设备并不宜包含 `set_config()` 函数。`set_config()` 函数的例子如代码清单 16.15 所示。

代码清单 16.15 网络设备驱动的 `set_config` 函数模板

```
1 int xxx_config(struct net_device *dev, struct ifmap *map)
2 {
3     if (netif_running(dev)) /* 不能设置一个正在运行状态的设备 */
4         return - EBUSY;
5
6     /* 假设不允许改变 I/O 地址 */
7     if (map->base_addr != dev->base_addr) {
8         printk(KERN_WARNING "xxx: Can't change I/O address\n");
9         return - EOPNOTSUPP;
10    }
11
12    /* 假设允许改变 IRQ */
13    if (map->irq != dev->irq)
14        dev->irq = map->irq;
15
16    return 0;
17 }
```

上述代码中的 `set_config()` 函数接受 IRQ 的修改，拒绝设备 I/O 地址的修改。具体的设备是否接收这些信息的修改，要视硬件的设计而定。

如果用户调用 `ioctl()` 时，命令类型在 `SIOCDEVPRIVATE` 和 `SIOCDEVPRIVATE+15` 之间，系统会调用驱动程序的 `do_ioctl()` 函数，进行设备专用数据的设置。这个设置大多数情况下也不需要。

驱动程序还应提供 `get_stats()` 函数用以向用户反馈设备状态和统计信息，该函数返回的是一个 `net_device_stats` 结构体，如代码清单 16.16 所示。

代码清单 16.16 网络设备驱动的 `get_stats` 函数模板

```
1 struct net_device_stats *xxx_stats(struct net_device *dev)
2 {
3     struct xxx_priv *priv = netdev_priv(dev);
4     return &priv->stats;
5 }
```

`net_device_stats` 结构体定义在内核的 `include/linux/netdevice.h` 文件中，它包含了比较完整的统计信息，如代码清单 16.17 所示。



代码清单 16.17 net_device_stats 结构体

```
1 struct net_device_stats
2 {
3     unsigned long rx_packets;      /* 收到的数据包数 */
4     unsigned long tx_packets;      /* 发送的数据包数 */
5     unsigned long rx_bytes;        /* 收到的字节数 */
6     unsigned long tx_bytes;        /* 发送的字节数 */
7     unsigned long rx_errors;        /* 收到的错误数据包数 */
8     unsigned long tx_errors;        /* 发生发送错误的数据包数 */
9     ...
10 };
```

上述代码清单只是列出了 `net_device_stats` 包含的主项目统计信息, 实际上, 这些项目还可以进一步细分, `net_device_stats` 中的其他信息给出了更详细的子项目统计, 详见 Linux 源代码。

`net_device_stats` 结构体适宜包含在设备的私有信息结构体中, 而其中统计信息的修改则应该在设备驱动的与发送和接收相关的具体函数中完成, 这些函数包括中断处理程序、数据包发送函数、数据包发送超时函数和数据包接收相关函数等。我们应该在这些函数中添加相应的代码, 如代码清单 16.18 所示。

代码清单 16.18 net_device_stats 结构体中统计信息的维护

```
1 /* 发送超时函数 */
2 void xxx_tx_timeout(struct net_device *dev)
3 {
4     struct xxx_priv *priv = netdev_priv(dev);
5     ...
6     priv->stats.tx_errors++; /* 发送错误包数加 1 */
7     ...
8 }
9
10 /* 中断处理函数 */
11 static void xxx_interrupt(int irq, void *dev_id)
12 {
13     switch (status & ISQ_EVENT_MASK) {
14         ...
15         case ISQ_TRANSMITTER_EVENT: /*
16             priv->stats.tx_packets++; /* 数据包发送成功, tx_packets 信息加 1 */
17             netif_wake_queue(dev); /* 通知上层协议 */
18             if ((status & (TX_OK | TX_LOST_CRD | TX_SQE_ERROR |
19                 TX_LATE_COL | TX_16_COL)) != TX_OK) { /* 读取硬件上的出错标志 */
20                 /* 根据错误的不同情况, 对 net_device_stats 的不同成员加 1 */
21                 if ((status & TX_OK) == 0)
22                     priv->stats.tx_errors++;
23                 if (status & TX_LOST_CRD)
24                     priv->stats.tx_carrier_errors++;
25                 if (status & TX_SQE_ERROR)
26                     priv->stats.tx_heartbeat_errors++;
27                 if (status & TX_LATE_COL)
28                     priv->stats.tx_window_errors++;
29                 if (status & TX_16_COL)
30                     priv->stats.tx_aborted_errors++;
31             }
32             break;
```

```

33 case ISQ_RX_MISS_EVENT:
34     priv->stats.rx_missed_errors += (status >> 6);
35     break;
36 case ISQ_TX_COL_EVENT:
37     priv->stats.collisions += (status >> 6);
38     break;
39 }
40 }

```

上述代码的第 6 行意味着在发送数据包超时，将发生发送错误的数据包数增 1。而第 13~38 行则意味着当网络设备中断产生时，中断处理程序读取硬件的相关信息以决定修改 `net_device_stats` 统计信息中的哪些项目和子项目，并将相应的项目增 1。

16.9 DM9000 网卡设备驱动实例

16.9.1 DM9000 网卡硬件描述

DM9000 是开发板采用的网络芯片，是一个高度集成而且功耗很低的高速网络控制器，可以和 CPU 直连，支持 10/100MB 以太网连接，芯片内部自带 4K 双字节的 SRAM（3KB 用来发送，13KB 用来接收）。针对不同的处理器，接口支持 8 位、16 位和 32 位。图 16.3 所示为 DM9000 的内部结构框架。

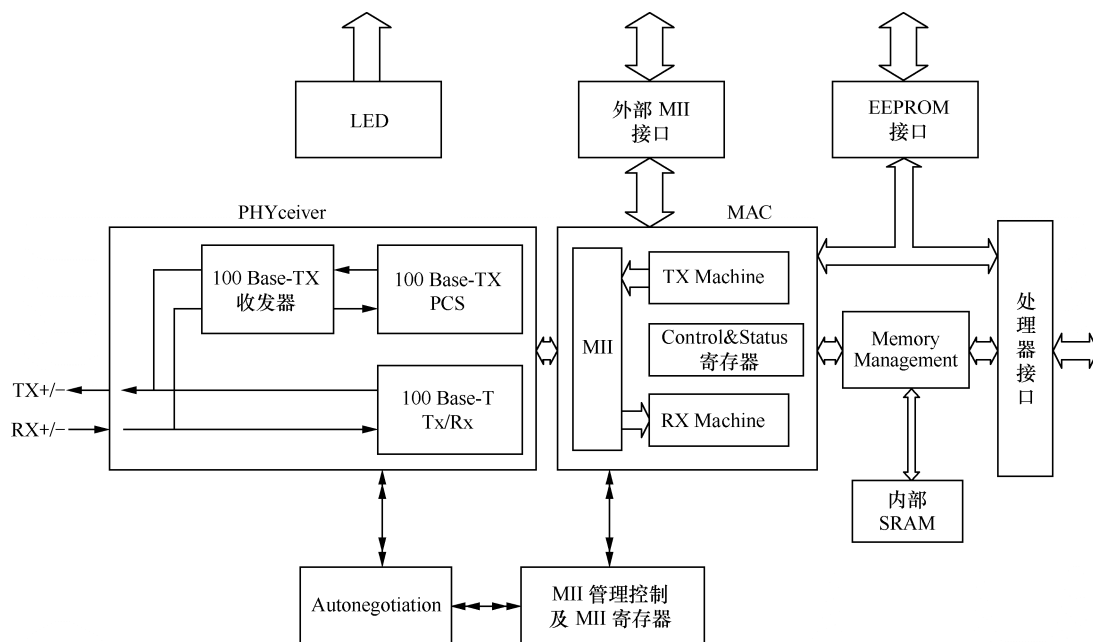
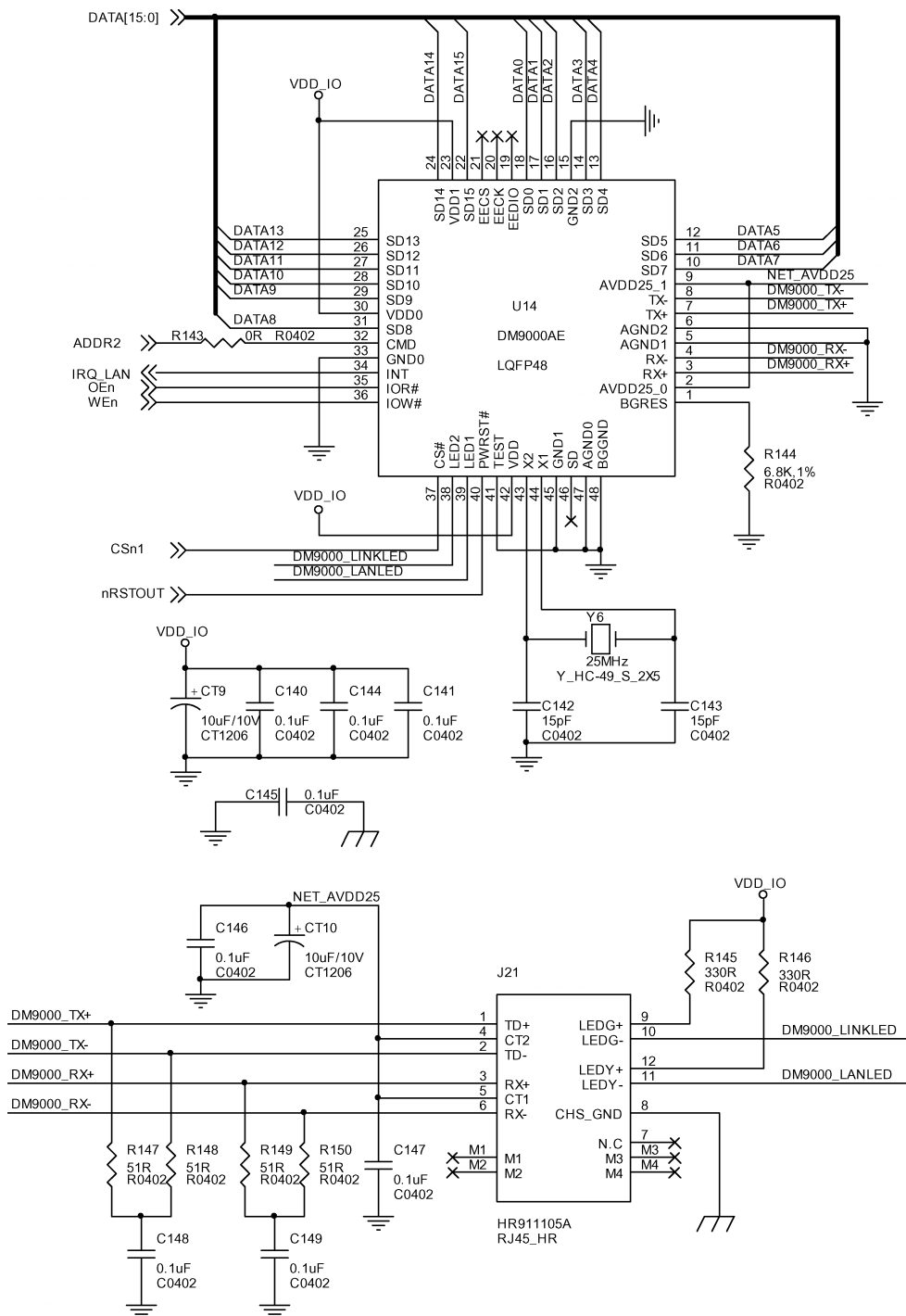


图 16.3 DM9000 以太网芯片的内部结构

在 LDD6410 开发板上，LDD6410 直接挂载在 S3C6410 的存储器总线上，其连接原理如图 16.4 所示。DM9000 占据 S3C6410 片选 1 的内存空间，而且由 S3C6410 地址线第 2 位驱动 CMD 引脚，



这说明书选 1 的地址 0~3 对应的是 DM9000 的 I/O 端口, 而 4~7 对应的是数据端口, 这一信息对于移植 DM9000 驱动到 LDD6410 非常关键。



ETHERNET

图 16.4 LDD6410 开发板的 DM9000 网卡连接原理

16.9.2 DM9000 网卡驱动设计分析

DM9000 网卡驱动位于内核源代码的 `drivers/net/dm9000.c`，它基于平台驱动架构，代码清单 16.19 抽取了它的主干。其核心工作是实现了前文所述 `net_device` 结构体中的 `hard_start_xmit`、`open`、`stop`、`set_multicast_list`、`do_ioctl`、`tx_timeout` 等成员函数并借助中断辅助进行网络数据包的收发，另外它也实现了 `ethtool_ops` 中的成员函数。

代码清单 16.19 DM9000 网卡驱动

```

1  /* Structure/enum 定义 ----- */
2  typedef struct board_info {
3
4      void __iomem *io_addr; /* 寄存器 I/O 基地址 */
5      void __iomem *io_data; /* 数据 I/O 基地址 */
6      u16      irq;          /* IRQ */
7
8      u16      tx_pkt_cnt;
9      ...
10 } board_info_t;
11
12 static int dm9000_ioctl(struct net_device *dev, struct ifreq *req, int cmd){...}
13
14 static const struct ethtool_ops dm9000_ethtool_ops = {
15     .get_drvinfo      = dm9000_get_drvinfo,
16     .get_settings     = dm9000_get_settings,
17     .set_settings     = dm9000_set_settings,
18     .get_msglevel     = dm9000_get_msglevel,
19     .set_msglevel     = dm9000_set_msglevel,
20     .nway_reset       = dm9000_nway_reset,
21     .get_link         = dm9000_get_link,
22     .get_eeprom_len   = dm9000_get_eeprom_len,
23     .get_eeprom       = dm9000_get_eeprom,
24     .set_eeprom       = dm9000_set_eeprom,
25 };
26
27 /* 设置 DM9000 多播地址 */
28 static void dm9000_hash_table(struct net_device *dev)
29
30 /* 看门狗超时，网络层将调用该函数 */
31 static void dm9000_timeout(struct net_device *dev)
32 {
33     ...
34     netif_stop_queue(dev);
35     netif_wake_queue(dev);
36     ...
37 }
38
39 static int dm9000_start_xmit(struct sk_buff *skb, struct net_device *dev)
40 {
41     ...
42     /* 将发送数据包移至 DM9000 的 TX RAM */
43     writeb(DM9000_MWCMD, db->io_addr);
44 
```



```
45 (db->outblk)(db->io_data, skb->data, skb->len);
46 dev->stats.tx_bytes += skb->len;
47 ...
48 }
49
50 /* 数据发送完成 */
51 static void dm9000_tx_done(struct net_device *dev, board_info_t *db)
52 {
53     ...
54     netif_wake_queue(dev);
55 }
56
57 /* 接收数据并传递给上层 */
58 static void
59 dm9000_rx(struct net_device *dev)
60 {
61     ...
62     netif_rx(skb);
63     dev->stats.rx_packets++;
64     ...
65 }
66
67 static irqreturn_t dm9000_interrupt(int irq, void *dev_id)
68 {
69     ...
70     return IRQ_HANDLED;
71 }
72
73 /* 打开网卡接口 */
74 static int dm9000_open(struct net_device *dev)
75 {
76     ..
77     netif_start_queue(dev);
78     ...
79     return 0;
80 }
81 /* 从 phyxcer 读一个 word */
82 static int dm9000_phy_read(struct net_device *dev, int phy_reg_unused, int reg){...};
83 /* 向 phyxcer 写一个 word */
84 static void
85 dm9000_phy_write(struct net_device *dev,
86                 int phyaddr_unused, int reg, int value){...}
87
88 static int __devinit
89 dm9000_probe(struct platform_device *pdev)
90 {
91     ...
92
93     /* Init network device */
94     ndev = alloc_etherdev(sizeof(struct board_info));
95
96     SET_NETDEV_DEV(ndev, &pdev->dev);
97
98     ether_setup(ndev);
99
```

```

100 ndev->open          = &dm9000_open;
101 ndev->hard_start_xmit = &dm9000_start_xmit;
102 ndev->tx_timeout     = &dm9000_timeout;
103 ndev->stop           = &dm9000_stop;
104 ndev->set_multicast_list = &dm9000_hash_table;
105 ndev->ethtool_ops    = &dm9000_ethtool_ops;
106 ndev->do_ioctl       = &dm9000_ioctl;
107
108 #ifdef CONFIG_NET_POLL_CONTROLLER
109 ndev->poll_controller = &dm9000_poll_controller;
110 #endif
111
112 db->msg_enable        = NETIF_MSG_LINK;
113 ...
114 db->mii.mdio_read     = dm9000_phy_read;
115 db->mii.mdio_write    = dm9000_phy_write;
116
117 platform_set_drvdata(pdev, ndev);
118 ret = register_netdev(ndev);
119
120 ...
121 }
122
123 static int __devexit
124 dm9000_drv_remove(struct platform_device *pdev)
125 {
126     struct net_device *ndev = platform_get_drvdata(pdev);
127
128     platform_set_drvdata(pdev, NULL);
129
130     unregister_netdev(ndev);
131     free_netdev(ndev);      /* free device structure */
132
133     return 0;
134 }
135
136 static struct platform_driver dm9000_driver = {
137     .driver = {
138         .name    = "dm9000",
139         .owner   = THIS_MODULE,
140     },
141     .probe      = dm9000_probe,
142     .remove     = __devexit_p(dm9000_drv_remove),
143 };
144
145 static int __init dm9000_init(void)
146 {
147     return platform_driver_register(&dm9000_driver);
148 }
149
150 static void __exit dm9000_cleanup(void)
151 {
152     platform_driver_unregister(&dm9000_driver);
153 }
154

```



```
155 module_init(dm9000_init);
156 module_exit(dm9000_cleanup);
```

DM9000 驱动的实现与具体 CPU 无关, 在将该驱动移植到特定电路板时, 只需要在板文件中为板上 DM9000 对应平台设备的寄存器和数据基地址进行赋值, 并指定正确的 IRQ 资源, 代码 16.20 给出了 LDD6410 开发板的板文件中对 DM9000 添加的内容。

代码清单 16.20 LDD6410 板文件中的 DM9000 的平台设备

```
1 static struct resource ldd6410_dm9000_resource[] = {
2     [0] = {
3         .start = 0x18000000,
4         .end   = 0x18000000 + 3,
5         .flags = IORESOURCE_MEM
6     },
7     [1] = {
8         .start = 0x18000000 + 0x4,
9         .end   = 0x18000000 + 0x7,
10        .flags = IORESOURCE_MEM
11    },
12    [2] = {
13        .start = IRQ_EINT(7),
14        .end   = IRQ_EINT(7),
15        .flags = IORESOURCE_IRQ | IORESOURCE_IRQ_HIGHLEVEL,
16    }
17 };
18
19
20 static struct dm9000_plat_data ldd6410_dm9000_platdata = {
21     .flags      = DM9000_PLATF_16BITONLY | DM9000_PLATF_NO_EEPROM,
22     .dev_addr   = { 0x0, 0x16, 0xd4, 0x9f, 0xed, 0xa4 },
23 };
24
25 static struct platform_device ldd6410_dm9000 = {
26     .name       = "dm9000",
27     .id         = 0,
28     .num_resources = ARRAY_SIZE(ldd6410_dm9000_resource),
29     .resource    = ldd6410_dm9000_resource,
30     .dev        = {
31         .platform_data = &ldd6410_dm9000_platdata,
32     }
33 };
```

另外, LDD6410 开发板的 U-BOOT 也全面支持 DM9000, 这使得我们可以在 U-BOOT 中运行网络命令或通过 tftp 下载 Linux 内核运行, 例如:

执行 ping:

```
LDD6410 # ping 192.168.1.111
dm9000 i/o: 0x18000300, id: 0x90000a46
MAC: 00:40:5c:26:0a:5b
operating at 100M full duplex mode host 192.168.1.111 is alive
LDD6410 #
```

下载 zImage:

```
LDD6410 # tftp 0xc0008000 zImage
dm9000 i/o: 0x18000300, id: 0x90000a46
```

```
MAC: 00:40:5c:26:0a:5b
operating at 100M full duplex mode
TFTP from server 192.168.1.111; our IP address is 192.168.1.20
Filename 'zImage'.
Load address: 0xc0008000
Loading: #####
#####
#####
#####
#####
#####
#####
#####
done
Bytes transferred = 2279916 (22c9ec hex)
```

启动内核:

```
LDD6410 # bootm 0xc0008000
Boot with zImage
Starting kernel ...
```

16.10 总结

Linux 网络设备驱动体系结构的层次化设计实现了对上层协议接口的统一和硬件驱动的对下层多样化硬件设备的可适应。程序员需要完成的工作集中在设备驱动功能层，网络设备接口层 `net_device` 结构体的存在将千变万化的网络设备得以抽象，使得设备功能层中除数据包接收以外的主体工作都由填充 `net_device` 的属性和函数指针完成。

在分析 `net_device` 数据结构的基础上，本章给出了设备驱动功能层设备初始化、数据包收发、打开和释放等函数的设计模板，这些模板对实际设备驱动的开发具有直接指导意义。有了这些模板，我们在设计具体设备的驱动时，不再需要关心程序的体系，而可以集中精力于硬件操作本身。

在 Linux 网络子系统和设备驱动中，套接字缓冲区 `sk_buff` 发挥着巨大的作用，是所有数据流动的载体。网络设备驱动和上层协议之间也依赖此结构进行数据包交互，因此，我们要特别牢记它的操作方法。

LINUX

第17章 Linux 音频设备驱动

在 Linux 系统中，先后出现了音频设备的 3 种框架：OSS、ALSA 和 ASoC，本节将在介绍数字音频设备及音频设备硬件接口的基础上讲解 OSS、ALSA 和 ASoC 驱动的结构。

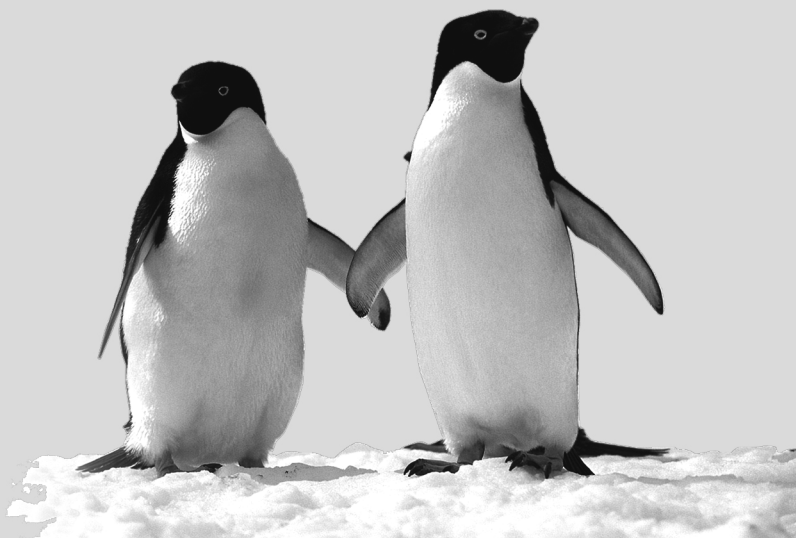
17.1~17.2 节讲解了音频设备及 PCM、IIS 和 AC'97 硬件接口。

17.3 节讲解了 Linux OSS 音频设备驱动的组成、mixer 接口、dsp 接口及用户空间编程方法。

17.4 节讲解了 Linux ALSA 音频设备驱动的组成、card 和组件管理、PCM 设备、control 接口、AC'97 API 及用户空间编程方法。

17.5 节讲解了 Linux ASoC 音频设备驱动的组成，Codec、CPU DAI 和板驱动。

17.6 节以 LDD6410 开发板上 S3C6410 通过 AC'97 接口外接 WM9714 的实例讲解了 ASoC 驱动。



17.1 数字音频设备

目前,手机、PDA、MP3 等许多嵌入式设备中包含了数字音频设备,一个典型的数字音频系统的电路组成如图 17.1 所示。图 17.1 中的嵌入式微控制器/DSP 中集成了 PCM、IIS 或 AC'97 音频接口,通过这些接口连接外部的音频编解码器即可实现声音的 AD 和 DA 转换,图中的功放完成模拟信号的放大功能。

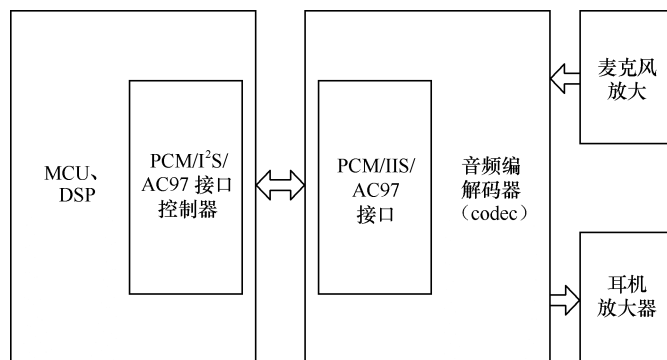


图 17.1 典型的数字音频系统电路

音频编解码器是数字音频系统的核心,衡量它的主要指标如下。

1. 采样频率

采样的过程就是将通常的模拟音频信号的电信号转换成二进制码 0 和 1 的过程,这些 0 和 1 便构成了数字音频文件。图 17.2 中的正弦曲线代表原始音频曲线,方格代表采样后得到的结果,两者越吻合说明采样结果越好。

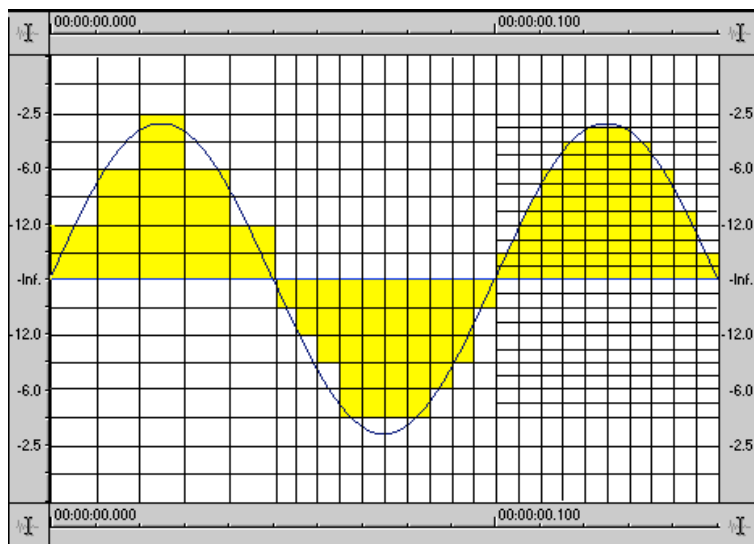


图 17.2 数字音频采样



采样频率是每秒钟的采样次数, 我们常说的 44.1kHz 采样频率就是每秒钟采样 44100 次。理论上采样频率越高, 转换精度越高, 目前主流的采样频率是 48kHz。

2. 量化精度

量化精度是指对采样数据分析的精度, 比如 24bit 量化精度就是指将标准电平信号按照 2 的 24 次方进行分析, 也就是说将图 17.2 中的纵坐标等分为 2^{24} 等分。量化精度越高, 声音就越逼真。

17.2 音频设备硬件接口

17.2.1 PCM 接口

针对不同的数字音频子系统, 出现了几种微处理器或 DSP 与音频器件间用于数字转换的接口。

最简单的音频接口是 PCM (脉冲编码调制) 接口, 该接口由时钟脉冲 (BCLK)、帧同步信号 (FS) 及接收数据 (DR) 和发送数据 (DX) 组成。在 FS 信号的上升沿, 数据传输从 MSB (Most Significant Bit) 开始, FS 频率等于采样率。FS 信号之后开始数据字的传输, 单个的数据位按顺序进行传输, 一个时钟周期传输一个数据字。

PCM 接口很容易实现, 原则上能够支持任何数据方案 and 任何采样率, 但需要每个音频通道获得一个独立的数据队列。

17.2.2 IIS 接口

IIS 接口 (Inter-IC Sound, 又称 I²S) 在 20 世纪 80 年代首先被 PHILIPS 用于消费音频产品, 并在一个称为 LRCLK (Left/Right CLOCK) 的信号机制中经过多路转换, 将两路音频信号变成单一的数据队列。当 LRCLK 为高时, 左声道数据被传输; LRCLK 为低时, 右声道数据被传输。与 PCM 相比, IIS 更适合于立体声系统。当然, IIS 的变体也支持多通道的时分复用, 因此可以支持多声道。

17.2.3 AC'97 接口

AC'97 (Audio Codec 1997) 是以 Intel 为首的 5 个 PC 厂商 Intel、Creative Labs、NS、Analog Device 与 Yamaha 共同提出的规格标准。与 PCM 和 IIS 不同, AC'97 不只是一种数据格式, 用于音频编码的内部架构规格, 它还具有控制功能。

AC'97 采用 AC-Link 与外部的编解码器相连, AC-Link 接口包括位时钟 (BITCLK)、同步信号校正 (SYNC) 和从编码到处理器及从处理器中解码 (SDATDIN 与 SDATAOUT) 的数据队列。AC'97 数据帧以 SYNC 脉冲开始, 包括 12 个 20 位时隙以及 1 个 16 位 “tag” 段, 共计 256 个数据序列。例如, 时序 “1” 和 “2” 用于访问编码的控制寄存器, 而时隙 “3” 和 “4” 分别负载左、右两个音频通道。“tag” 段表示其他时隙中哪一个包含有效数据。把帧分成时隙使传输控制信号和音频数据仅通过 4 根线到达 9 个音频通道或转换成其他数据流成为可能。图 17.3 所示 AC'97 的时序图。

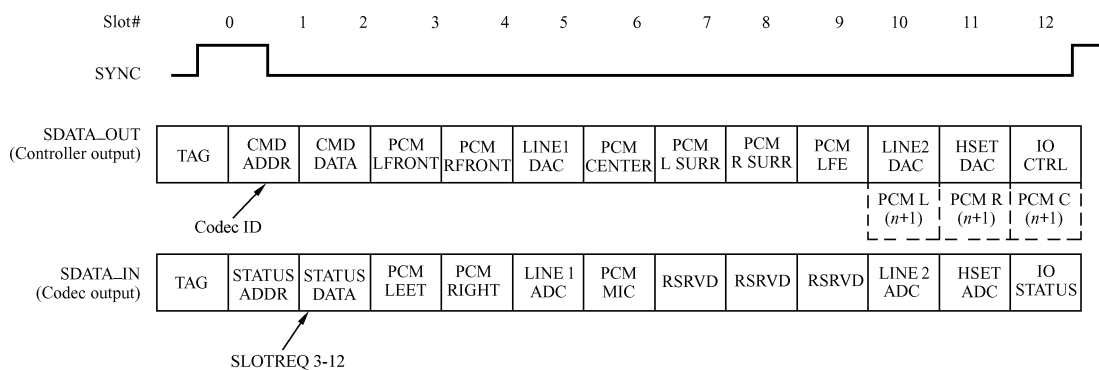


图 17.3 AC'97 接口时序

PCM、IIS 和 AC'97 各有其优点和应用范围，例如在 CD、MD、MP3 随身听多采用 IIS 接口，移动电话多采用 PCM 接口，智能手机、PDA 则多使用和 PC 一样的 AC'97 编码格式。

17.3 Linux OSS 音频设备驱动

17.3.1 OSS 驱动的组成

OSS 标准中有两个最基本的音频设备：**mixer**（混音器）和 **dsp**（数字信号处理器）。

在声卡的硬件电路中，**mixer** 是一个很重要的组成部分，它的作用是将多个信号组合或者叠加在一起，对于不同的声卡来说，其混音器的作用可能各不相同。OSS 驱动中，`/dev/mixer` 设备文件是应用程序对 **mixer** 进行操作的软件接口。

混音器电路通常由两部分组成：输入混音器（**input mixer**）和输出混音器（**output mixer**）。输入混音器负责从多个不同的信号源接收模拟信号，这些信号源有时也被称为混音通道或者混音设备。模拟信号通过增益控制器和由软件控制的音量调节器，在不同的混音通道中进行级别（**level**）调制，然后被送到输入混音器中进行声音的合成。混音器上的电子开关可以控制哪些通道中有信号与混音器相连，有些声卡只允许连接一个混音通道作为录音的音源，而有些声卡则允许对混音通道做任意的连接。经过输入混音器处理后的信号仍然为模拟信号，它们将被送到 A/D 转换器进行数字化处理。

输出混音器的工作原理与输入混音器类似，同样也有多个信号源与混音器相连，并且事先都经过了增益调节。当输出混音器对所有的模拟信号进行了混合之后，通常还会有一个总控增益调节器来控制输出声音的大小，此外还有一些音调控制器来调节输出声音的音调。经过输出混音器处理后的信号也是模拟信号，它们最终会被送给喇叭或者其他的模拟输出设备。

对混音器的编程包括如何设置增益控制器的级别，以及怎样在不同的音源间进行切换，这些操作通常来讲是不连续的，而且不会像录音或者播放那样需要占用大量的计算机资源。由于混音器的操作不符合典型的读/写操作模式，因此除了 `open()` 和 `close()` 这两个系统调用之外，大部分的操作都是通过 `ioctl()` 系统调用来完成的。与 `/dev/dsp` 不同，`/dev/mixer` 允许多个应用程序同时访问，并且混音器的设置值会一直保持到对应的设备文件被关闭为止。



DSP 也称为编解码器, 实现录音和放音, 其对应的设备文件是 `/dev/dsp` 或 `/dev/sound/dsp`。OSS 声卡驱动程序提供的 `/dev/dsp` 是用于数字采样和数字录音的设备文件, 向该设备写数据即意味着激活声卡上的 D/A 转换器进行播放, 而向该设备读数据则意味着激活声卡上的 A/D 转换器进行录音。

从 DSP 设备读取数据时, 从声卡输入的模拟信号经过 A/D 转换器变成数字采样后的样本, 保存在声卡驱动程序的内核缓冲区中, 当应用程序通过 `read()` 系统调用从声卡读取数据时, 保存在内核缓冲区中的数字采样结果将被复制到应用程序所指定的用户缓冲区中。如果应用程序读取数据的速度过慢, 以致低于声卡的采样频率, 那么多余的数据将会被丢弃 (即 `overflow`); 如果读取数据的速度过快, 以致高于声卡的采样频率, 那么声卡驱动程序将会阻塞那些请求数据的应用程序, 直到新的数据到来为止。

向 DSP 设备写入数据时, 数字信号会经过 D/A 转换器变成模拟信号, 然后产生声音。应用程序写入数据的速度应该至少等于声卡的采样频率, 过慢会产生声音暂停或者停顿的现象 (即 `underflow`)。如果用户写入过快的话, 它会被内核中的声卡驱动程序阻塞, 直到硬件有能力处理新的数据为止。

与其他设备有所不同, 声卡通常不需要支持非阻塞 (`non-blocking`) 的 I/O 操作。即便内核 OSS 驱动提供了非阻塞的 I/O 支持, 用户空间也很少采用。

无论是从声卡读取数据, 或是向声卡写入数据, 事实上都具有特定的格式 (`format`), 如无符号 8 位、单声道、8kHz 采样率, 如果默认值无法达到要求, 可以通过 `ioctl()` 系统调用来改变它们。通常说来, 在应用程序中打开设备文件 `/dev/dsp` 之后, 接下去就应该为其设置恰当的格式, 然后才能从声卡读取或者写入数据。

17.3.2 mixer 接口

```
int register_sound_mixer(struct file_operations *fops, int dev);
```

上述函数用于注册一个混音器, 第一个参数 `fops` 即是文件操作接口, 第二个参数 `dev` 是设备编号, 如果填入 -1, 则系统自动分配一个设备编号。`mixer` 是一个典型的字符设备, 因此编码的主要工作是实现 `file_operations` 中的 `open()`、`ioctl()` 等函数。

`mixer` 接口 `file_operations` 中的最重要函数是 `ioctl()`, 它实现混音器的不同 I/O 控制命令, 代码清单 17.1 所示为一个 `ioctl()` 的范例。

代码清单 17.1 mixer()接口的 ioctl()函数范例

```
1 static int mixdev_ioctl(struct inode *inode, struct file *file, unsigned int cmd,
2 unsigned long arg)
3 {
4     ...
5     switch (cmd) {
6         case SOUND_MIXER_READ_MIC:
7             ...
8         case SOUND_MIXER_WRITE_MIC:
9             ...
10        case SOUND_MIXER_WRITE_RECSRC:
11            ...
12        case SOUND_MIXER_WRITE_MUTE:
13            ...
14    }
```

```

14  /* 其他命令 */
15  return mixer_ioctl(codec, cmd, arg);
16 }

```

17.3.3 dsp 接口

```
int register_sound_dsp(struct file_operations *fops, int dev);
```

上述函数与 `register_sound_mixer()` 类似，它用于注册一个 dsp 设备，第一个参数 `fops` 即是文件操作接口，第二个参数 `dev` 是设备编号，如果填入 -1，则系统自动分配一个设备编号。dsp 也是一个典型的字符设备，因此编码的主要工作是实现 `file_operations` 中的 `read()`、`write()`、`ioctl()` 等函数。

dsp 接口 `file_operations` 中的 `read()` 和 `write()` 函数非常重要，`read()` 函数从音频控制器中获取录音数据到缓冲区并复制到用户空间，`write()` 函数从用户空间复制音频数据到内核空间缓冲区并最终发送到音频控制器。

dsp 接口 `file_operations` 中的 `ioctl()` 函数处理对采样率、量化精度、DMA 缓冲区块大小等参数设置 I/O 控制命令的处理。

在数据从缓冲区复制到音频控制器的过程中，通常会使用 DMA，DMA 对声卡而言非常重要。例如，在放音时，驱动设置完 DMA 控制器的源数据地址（内存中的 DMA 缓冲区）、目的地址（音频控制器 FIFO）和 DMA 的数据长度，DMA 控制器会自动发送缓冲区的数据填充 FIFO，直到发送完相应的数据长度后才中断一次。

在 OSS 驱动中，建立存放音频数据的环形缓冲区（ring buffer）通常是值得推荐的方法。此外，在 OSS 驱动中，一般会将一个较大的 DMA 缓冲区分成若干个大小相同的块（这些块也被称为“段”，即 fragment），驱动程序使用 DMA 每次在声音缓冲区和声卡之间搬移一个 fragment。在用户空间，可以使用 `ioctl()` 系统调用来调整块的大小和个数。

除了 `read()`、`write()` 和 `ioctl()` 外，dsp 接口的 `poll()` 函数通常也需要被实现，以向用户反馈目前能否读写 DMA 缓冲区。

在 OSS 驱动初始化过程中，会调用 `register_sound_dsp()` 和 `register_sound_mixer()` 注册 dsp 和 mixer 设备；在模块卸载的时候，会调用代码清单 17.2。

代码清单 17.2 OSS 驱动初始化注册 dsp 和 mixer 设备

```

1  static int xxx_init(void)
2  {
3      struct xxx_state *s = &xxx_state;
4      ...
5      /* 注册 dsp 设备 */
6      if ((audio_dev_dsp = register_sound_dsp(&xxx_audio_fops, - 1)) < 0)
7          goto err_dev1;
8      /* 设备 mixer 设备 */
9      if ((audio_dev_mixer = register_sound_mixer(&xxx_mixer_fops, - 1)) < 0)
10         goto err_dev2;
11     ...
12 }
13
14 void __exit xxx_exit(void)
15 {

```



```
16 /* 注销 dsp 和 mixer 设备接口 */
17 unregister_sound_dsp(audio_dev_dsp);
18 unregister_sound_mixer(audio_dev_mixer);
19 ...
20 }
```

根据 17.3.2 和 17.3.3 小节的分析, 可以画出一个 Linux OSS 驱动结构的简图, 如图 17.4 所示。

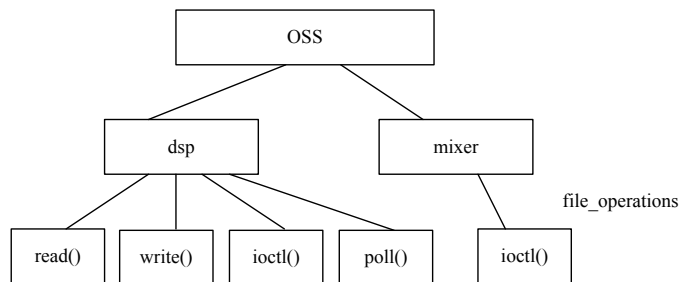


图 17.4 Linux OSS 驱动结构

17.3.4 OSS 用户空间编程

1. dsp 编程

对 OSS 驱动声卡的编程使用 Linux 文件接口函数, 如图 17.5 所示, dsp 接口的操作一般包括如下几个步骤。

(1) 打开设备文件/dev/dsp。

采用何种模式对声卡进行操作也必须在打开设备时指定, 对于不支持全双工的声卡来说, 应该使用只读或者只写的方式打开, 只有那些支持全双工的声卡, 才能以读写的方式打开, 这还依赖于驱动程序的具体实现。Linux 允许应用程序多次打开或者关闭与声卡对应的设备文件, 从而能够很方便地在放音状态和录音状态之间进行切换。

(2) 如果有需要, 设置缓冲区大小。

运行在 Linux 内核中的声卡驱动程序专门维护了一个缓冲区, 其大小会影响到播放和录音时的效果, 使用 `ioctl()` 系统调用可以对它的尺寸进行恰当设置。调节驱动程序中缓冲区大小的操作不是必须的, 如果没有特殊的要求, 一般采用默认的缓冲区大小也就可以了。如果想设置缓冲区的大小, 则通常应紧跟在设备文件打开之后, 这是因为对声卡的其他操作有可能会驱动导致驱动程序无法再修改其缓冲区的大小。

(3) 设置声道 (channel) 数量。

根据硬件设备和驱动程序的具体情况, 可以设置为单声道或者立体声。

(4) 设置采样格式和采样频率

采样格式包括 `AFMT_U8` (无符号 8 位)、`AFMT_S8` (有符号 8 位)、`AFMT_U16_LE` (小端模式, 无符号 16 位)、`AFMT_U16_BE` (大端模式, 无符号 16 位)、`AFMT_MPEG`、`AFMT_AC3` 等。使用 `SNDCTL_DSP_SETFMT` IO 控制命令可以设置采样格式。

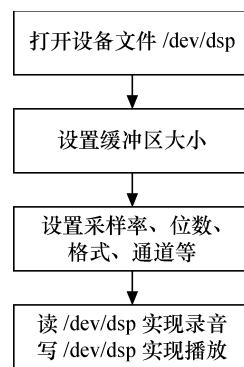


图 17.5 OSS dsp 接口用户空间操作流程

对于大多数声卡来说，其支持的采样频率范围一般为 5kHz~44.1kHz 或者 48kHz，但并不意味着该范围内的所有连续频率都会被硬件支持，在 Linux 系统下进行音频编程时最常用到的几种采样频率是 11025Hz、16000Hz、22050Hz、32000Hz 和 44100Hz。使用 SNDCTL_DSP_SPEED IO 控制命令可以设置采样频率。

(5) 读写/dev/dsp 实现播放或录音。

代码清单 17.3 (read、write、ioctl 的出错处理没有列出) 的程序实现了利用/dev/dsp 接口进行声音录制和播放的过程，它的功能是先录制几秒钟音频数据，将其存放在内存缓冲区中，然后再进行播放。

代码清单 17.3 OSS dsp 接口应用编程范例

```
1 #include ...
2 #define LENGTH 3          /* 存储秒数 */
3 #define RATE 8000         /* 采样频率 */
4 #define SIZE 8            /* 量化位数 */
5 #define CHANNELS 1        /* 声道数目 */
6 /* 用于保存数字音频数据的内存缓冲区 */
7 unsigned char buf[LENGTH * RATE * SIZE * CHANNELS / 8];
8 int main()
9 {
10  int fd; /* 声音设备的文件描述符 */
11  int arg; /* 用于 ioctl 调用的参数 */
12  int status; /* 系统调用的返回值 */
13  /* 打开声音设备 */
14  fd = open("/dev/dsp", O_RDWR);
15
16  /* 设置采样时的量化位数 */
17  arg = SIZE;
18  status = ioctl(fd, SOUND_PCM_WRITE_BITS, &arg);
19
20  /* 设置采样时的通道数目 */
21  arg = CHANNELS;
22  status = ioctl(fd, SOUND_PCM_WRITE_CHANNELS, &arg);
23
24  /* 设置采样率 */
25  arg = RATE;
26  status = ioctl(fd, SOUND_PCM_WRITE_RATE, &arg);
27
28  /* 循环，直到按下 [Ctrl+C] */
29  while (1) {
30    printf("Say something:\n");
31    status = read(fd, buf, sizeof(buf)); /* 录音 */
32
33    printf("You said:\n");
34    status = write(fd, buf, sizeof(buf)); /* 放音 */
35
36    /* 在继续录音前等待放音结束 */
37    status = ioctl(fd, SOUND_PCM_SYNC, 0);
38  }
39 }
40 }
```



2. mixer 编程

声卡上的混音器由多个混音通道组成，它们可以通过驱动程序提供的设备文件/dev/mixer 进行编程。对混音器的操作一般都通过 ioctl()系统调用来完成，所有控制命令都以 SOUND_MIXER 或者 MIXER 开头，表 17.1 列出了常用的混音器控制命令。

表 17.1 混音器常用命令

命 令	作 用
SOUND_MIXER_VOLUME	主音量调节
SOUND_MIXER_BASS	低音控制
SOUND_MIXER_TREBLE	高音控制
SOUND_MIXER_SYNTH	FM 合成器
SOUND_MIXER_PCM	主 D/A 转换器
SOUND_MIXER_SPEAKER	PC 喇叭
SOUND_MIXER_LINE	音频线输入
SOUND_MIXER_MIC	麦克风输入
SOUND_MIXER_CD	CD 输入
SOUND_MIXER_IMIX	收音音量
SOUND_MIXER_ALTPCM	从 D/A 转换器
SOUND_MIXER_RECLEV	录音音量
SOUND_MIXER_IGAIN	输入增益
SOUND_MIXER_OGAIN	输出增益
SOUND_MIXER_LINE1	声卡的第 1 输入
SOUND_MIXER_LINE2	声卡的第 2 输入
SOUND_MIXER_LINE3	声卡的第 3 输入

对声卡的输入增益和输出增益进行调节是混音器的一个主要作用，目前大部分声卡采用的是 8 位或者 16 位的增益控制器，声卡驱动程序会将它们变换成百分比的形式，也就是说无论是输入增益还是输出增益，其取值范围都是从 0~100。

(1) SOUND_MIXER_READ 宏。

在进行混音器编程时，可以使用 SOUND_MIXER_READ 宏来读取混音通道的增益大小，例如，如下代码可以获得麦克风的输入增益：

```
ioctl(fd, SOUND_MIXER_READ(SOUND_MIXER_MIC), &vol);
```

对于只有一个混音通道的单声道设备来说，返回的增益大小保存在低位字节中。而对于支持多个混音通道的双声道设备来说，返回的增益大小实际上包括两个部分，分别代表左、右两个声道的值，其中低位字节保存左声道的音量，而高位字节则保存右声道的音量。下面的代码可以从返回值中依次提取左右声道的增益大小：

```
int left, right;
left = vol & 0xff;
right = (vol & 0xff00) >> 8;
```

(2) SOUND_MIXER_WRITE 宏。

如果想设置混音通道的增益大小,则可以通过 SOUND_MIXER_WRITE 宏来实现,例如下面的语句可以用来设置麦克风的输入增益:

```
vol = (right << 8) + left;
ioctl(fd, SOUND_MIXER_WRITE(SOUND_MIXER_MIC), &vol);
```

(3) 查询 MIXER 信息。

声卡驱动程序提供了多个 ioctl() 系统调用来获得混音器的信息,它们通常返回一个整型的位掩码,其中每一位分别代表一个特定的混音通道,如果相应的位为 1,则说明与之对应的混音通道是可用的。

通过 SOUND_MIXER_READ_DEVMASK 返回的位掩码查询出能够被声卡支持的每一个混音通道,而通过 SOUND_MIXER_READ_RECMASK 返回的位掩码则可以查询出能够被当作录音源的每一个通道。例如,如下代码可用来检查 CD 输入是否是一个有效的混音通道:

```
ioctl(fd, SOUND_MIXER_READ_DEVMASK, &devmask);
if (devmask & SOUND_MIXER_CD)
    printf("The CD input is supported");
```

如下代码可用来检查 CD 输入是否是一个有效的录音源:

```
ioctl(fd, SOUND_MIXER_READ_RECMASK, &recmask);
if (recmask & SOUND_MIXER_CD)
    printf("The CD input can be a recording source");
```

大多数声卡提供了多个录音源,通过 SOUND_MIXER_READ_RECSRC 可以查询出当前正在使用的录音源,同一时刻可使用两个或两个以上的录音源,具体由声卡硬件本身决定。相应地,使用 SOUND_MIXER_WRITE_RECSRC 可以设置声卡当前使用的录音源,如下代码可以将 CD 输入作为声卡的录音源使用。

```
devmask = SOUND_MIXER_CD;
ioctl(fd, SOUND_MIXER_WRITE_RECSRC, &devmask);
```

此外,所有的混音通道都有单声道和双声道的区别,如果需要知道哪些混音通道提供了对立体声的支持,可以通过 SOUND_MIXER_READ_STEREODEVS 来获得。

代码清单 17.4 的程序实现了利用/dev/mixer 接口对混音器进行编程的过程,该程序可对各种混音通道的增益进行调节。

代码清单 17.4 OSS mixer 接口应用编程范例

```
1 #include ...
2 /* 用来存储所有可用混音设备的名称 */
3 const char *sound_device_names[] = SOUND_DEVICE_NAMES;
4 int fd; /* 混音设备所对应的文件描述符 */
5 int devmask, stereodevs; /* 混音器信息对应的 bit 掩码 */
6 char *name;
7 /* 显示命令的使用方法及所有可用的混音设备 */
8 void usage()
9 {
10     int i;
11     fprintf(stderr, "usage: %s <device> <left-gain%> <right-gain%>\n"
12         "%s <device> <gain%>\n\n" "Where <device> is one of:\n", name, name);
13     for(i = 0; i < SOUND_MIXER_NRDEVICES; i++)
14         if ((1 << i) & devmask)
15         /* 只显示有效的混音设备 */
```



```
16     fprintf(stderr, "%s ", sound_device_names[i]);
17     fprintf(stderr, "\n");
18     exit(1);
19 }
20
21 int main(int argc, char *argv[])
22 {
23     int left, right, level; /* 增益设置 */
24     int status; /* 系统调用的返回值 */
25     int device; /* 选用的混音设备 */
26     char *dev; /* 混音设备的名称 */
27     int i;
28     name = argv[0];
29     /* 以只读方式打开混音设备 */
30     fd = open("/dev/mixer", O_RDONLY);
31     if (fd == -1) {
32         perror("unable to open /dev/mixer");
33         exit(1);
34     }
35
36     /* 获得所需要的信息 */
37     status = ioctl(fd, SOUND_MIXER_READ_DEVMASK, &devmask);
38     if(status == -1)
39         perror("SOUND_MIXER_READ_DEVMASK ioctl failed");
40     status = ioctl(fd, SOUND_MIXER_READ_STEREODEVS, &stereodevs);
41     if (status == -1)
42         perror("SOUND_MIXER_READ_STEREODEVS ioctl failed");
43     /* 检查用户输入 */
44     if (argc != 3 && argc != 4)
45         usage();
46     /* 保存用户输入的混音器名称 */
47     dev = argv[1];
48     /* 确定即将用到的混音设备 */
49     for (i = 0; i < SOUND_MIXER_NRDEVICES; i++)
50         if ((1 << i) & devmask) && !strcmp(dev, sound_device_names[i]))
51             break;
52     if (i == SOUND_MIXER_NRDEVICES) {
53         /* 没有找到匹配项 */
54         fprintf(stderr, "%s is not a valid mixer device\n", dev);
55         usage();
56     }
57     /* 查找到有效的混音设备 */
58     device = i;
59     /* 获取增益值 */
60     if (argc == 4) {
61         /* 左、右声道均给定 */
62         left = atoi(argv[2]);
63         right = atoi(argv[3]);
64     } else {
65         /* 左、右声道设为相等 */
66         left = atoi(argv[2]);
67         right = atoi(argv[2]);
68     }
69
70     /* 对非立体声设备给出警告信息 */
```



```

71 if ((left != right) && !((1 << i) &stereodevs))
72     fprintf(stderr, "warning: %s is not a stereo device\n", dev);
73
74 /* 将两个声道的值合到同一变量中 */
75 level = (right << 8) + left;
76
77 /* 设置增益 */
78 status = ioctl(fd, MIXER_WRITE(device), &level);
79 if (status == - 1) {
80     perror("MIXER_WRITE ioctl failed");
81     exit(1);
82 }
83 /* 获得从驱动返回的左右声道的增益 */
84 left = level &0xff;
85 right = (level &0xff00) >> 8;
86 /* 显示实际设置的增益 */
87 fprintf(stderr, "%s gain set to %d%% / %d%%\n", dev, left, right);
88 /* 关闭混音设备 */
89 close(fd);
90 return 0;
91 }

```

编译上述程序为可执行文件 mixer，执行 ./mixer <device> <left-gain%> <right-gain%> 或 ./mixer <device> <gain%> 可设置增益，device 可以是 vol、pcm、speaker、line、mic、cd、igain、line1、phin、video。

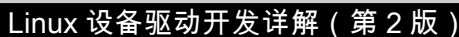
17.4 Linux ALSA 音频设备驱动

17.4.1 ALSA 的组成

虽然 OSS 已经非常成熟，但它毕竟是一个没有完全开放源代码的商业产品，而且目前基本上在 Linux mainline 中失去了更新。而 ALSA（Advanced Linux Sound Architecture）恰好弥补了这一空白，它符合 GPL，是在 Linux 下进行音频编程时另一种可供选择的声卡驱动体系结构。ALSA 除了像 OSS 那样提供了一组内核驱动程序模块之外，还专门为简化应用程序的编写提供了相应的函数库，与 OSS 提供的基于 ioctl 的原始编程接口相比，ALSA 函数库使用起来要更加方便一些。ALSA 的主要特点如下。

- 支持多种声卡设备。
- 模块化的内核驱动程序。
- 支持 SMP 和多线程。
- 提供应用开发函数库（alsa-lib）以简化应用程序开发。
- 支持 OSS API，兼容 OSS 应用程序。

ALSA 具有更加友好的编程接口，并且完全兼容于 OSS，对应用程序员来讲无疑是一个更佳的选择。ALSA 系统包括驱动包 alsa-driver、开发包 alsa-lib、开发包插件 alsa-libplugins、设置管理工具包 alsa-utils、其他声音相关处理小程序包 alsa-tools、特殊音频固件支持包 alsa-firmware、



`alsa-driver` 指内核驱动程序，包括硬件相关的代码和一些公共代码，非常庞大，代码总量达数十万行；`alsa-libs` 指用户空间的函数库，提供给应用程序使用，应用程序应包含头文件 `asoundlib.h`，并使用共享库 `libasound.so`；`alsa-utils` 包含一些基于 ALSA 的用于控制声卡的应用程序，如 `alsacnf`（侦测系统中声卡并写一个适合的 ALSA 配置文件）、`alsactl`（控制 ALSA 声卡驱动的高级设置）、`alsamixer`（基于 `ncurses` 的混音器程序）、`amidi`（用于读写 ALSA RawMIDI）、`amixer`（ALSA 声卡混音器的命令行控制）、`aplay`（基于命令行的声音文件播放）、`arecord`（基于命令行的声音文件录制）等。

- 信息接口 (Information Interface, /proc/asound);
- 控制接口 (Control Interface, /dev/snd/controlCX);
- 混音器接口 (Mixer Interface, /dev/snd/mixerCXDX);
- PCM 接口 (PCM Interface, /dev/snd/pcmCXDX);
- Raw 迷笛接口 (Raw MIDI Interface, /dev/snd/midiCXDX);
- 音序器接口 (Sequencer Interface, /dev/snd/seq);
- 定时器接口 (Timer Interface, /dev/snd/timer)。

```

graph TD
    subgraph OSS_Stack [OSS Architecture]
        direction TB
        OSSApp1[OSS应用] --> OSSAPI1[OSS API]
        OSSAPI1 --> LD_PRE_LOAD[LD_PRE_LOAD]
        LD_PRE_LOAD --> OSSUserSim[OSS用户空间模拟]
        OSSUserSim --> OSSAPI2[OSS API]
        OSSAPI2 --> OSSModSim[OSS模拟模块]
    end

    subgraph ALSA_Stack [ALSA Architecture]
        direction TB
        ALSAApp[ALSA应用] --> ALSALibAPI[ALSA库API]
        ALSALibAPI --> ALSALib[ALSA库]
        ALSALib --> HWAccess[硬件访问]
        HWAccess --> Plugins[插件 Conversion, routing等]
        Plugins --> ALSAKernelAPI[ALSA内核API PCM/MIDI/Control/Sequencer]
        ALSAKernelAPI --> ALSADriver[ALSA驱动]
    end

    OSSApp1 --> ALSAApp
    OSSAPI1 --> ALSALibAPI
    LD_PRE_LOAD --> ALSALib
    OSSUserSim --> HWAccess
    OSSAPI2 --> ALSAKernelAPI
    OSSModSim --> ALSADriver
    ALSADriver --> HW[硬件]

```

图 17.6 ALSA 体系结构

图 17.6 所示为 ALSA 声卡驱动与用户空间体系结构的简图，从中可以看出 ALSA 内核驱动与用户空间库及 OSS 之间的关系。

17.4.2 card 和组件管理

对于每个声卡而言，必须创建一个 `card` 实例。`card` 是声卡的“总部”，它管理这个声卡上的所有设备（组件），如 PCM、mixers、MIDI、synthesizer 等。因此，`card` 和组件是 ALSA 声卡驱动中的主要组成元素。

[illegible]

idx 是 card 索引号, xid 是标识字符串, module 一般为 THIS_MODULE, extra_size 是要分配的额外数据的大小, 分配的 extra size 大小的内存将作为 card->private data。

2. 创建组件

```
int snd_device_new(struct snd_card *card, snd_device_type_t type,
                  void *device_data, struct snd_device_ops *ops);
```

当 card 被创建后, 设备 (组件) 能够被创建并关联于该 card。第 1 个参数是 snd_card_new() 创建的 card 指针, 第 2 个参数 type 指的是 device-level 即设备类型, 形式为 SNDRV_DEV_XXX, 包括 SNDRV_DEV_CODEC、SNDRV_DEV_CONTROL、SNDRV_DEV_PCM、SNDRV_DEV_RAWMIDI 等, 用户自定义设备的 device-level 是 SNDRV_DEV_LOWLEVEL, ops 参数是 1 个函数集 (定义为 snd_device_ops 结构体) 的指针, device_data 是设备数据指针, 注意函数 snd_device_new() 本身不会分配设备数据的内存, 因此应事先分配。

3. 组件释放

每个 ALSA 预定义的组件在构造时需调用 snd_device_new(), 而每个组件的析构方法则在函数集中被包含。对于 PCM、AC97 此类预定义组件, 我们不需关心它们的析构, 而对于自定义的组件, 则需要填充 snd_device_ops 中的析构函数指针 dev_free, 这样, 当 snd_card_free() 被调用时, 组件将自动被释放。

4. 芯片特定的数据 (Chip-Specific Data)

芯片特定的数据一般以 struct xxxchip 结构体形式组织, 这个结构体中包含芯片相关的 I/O 端口地址、资源指针、中断号等, 其意义等同于字符设备驱动中的 file->private_data。

定义芯片特定的数据主要有两种方法, 一种方法是将 sizeof(struct xxxchip) 传入 snd_card_new() 作为 extra_size 参数, 它将自动成为 snd_card 的 private_data 成员, 如代码清单 17.5 所示; 另一种方法是在 snd_card_new() 传入给 extra_size 参数 0, 再分配 sizeof(struct xxxchip) 的内存, 将分配内存的地址传入 snd_device_new() 的 device_data 的参数, 如代码清单 17.6 所示。

代码清单 17.5 创建芯片特定的数据方法 1

```
1 struct xxxchip { /* 芯片特定的数据结构体 */
2     ...
3 };
4 card = snd_card_new(index, id, THIS_MODULE, sizeof(struct
5 xxxchip)); /* 创建声卡并申请 xxxchip 内存作为 card-> private_data */
6 struct xxxchip *chip = card->private_data;
```

代码清单 17.6 创建芯片特定的数据方法 2

```
1 struct snd_card *card;
2 struct xxxchip *chip;
3 /* 使用 0 作为第 4 个参数, 并动态分配 xxx_chip 的内存*/
4 card = snd_card_new(index[dev], id[dev], THIS_MODULE, 0);
5 ...
6 chip = kzalloc(sizeof(*chip), GFP_KERNEL);
7 /* 在 xxxchip 结构体中, 应该包括声卡指针*/
8 struct xxxchip {
9     struct snd_card *card;
10    ...
11 };
12 /* 并将其 card 成员赋值为 snd_card_new() 创建的 card 指针*/
13 chip->card = card;
14 static struct snd_device_ops ops = {
15     . dev_free = snd_xxx_chip_dev_free, /* 组件析构*/
```



```
16 };
17 ...
18 /* 创建自定义组件*/
19 snd_device_new(card, SNDRV_DEV_LOWLEVEL, chip, &ops);
20 /* 在析构函数中释放 xxxchip 内存*/
21 static int snd_xxx_chip_dev_free(struct snd_device *device)
22 {
23     return snd_xxx_chip_free(device->device_data); /* 释放*/
24 }
```

5. 注册/释放声卡

当 `snd_card` 被准备好以后, 可使用 `snd_card_register()` 函数注册这个声卡, 如下所示:

```
int snd_card_register(struct snd_card *card)
```

对应的 `snd_card_free()` 完成相反的功能, 如下所示:

```
int snd_card_free(struct snd_card *card);
```

17.4.3 PCM 设备

每个声卡最多可以有 4 个 PCM 实例, 一个 PCM 实例对应一个设备文件。PCM 实例由 PCM 播放和录音流组成, 而每个 PCM 流又由一个或多个 PCM 子流组成。有的声卡支持多重播放功能, 例如, `emul0k1` 包含一个有 32 个立体声子流的 PCM 播放设备。

1. PCM 实例构造

```
int snd_pcm_new(struct snd_card *card, char *id, int device,
               int playback_count, int capture_count, struct snd_pcm ** rpcm);
```

第 1 个参数是 `card` 指针, 第 2 个是标识字符串, 第 3 个是 PCM 设备索引 (0 表示第 1 个 PCM 设备), 第 4 和第 5 个分别为播放和录音设备的子流数。当存在多个子流时, 需要恰当地处理 `open()`、`close()` 和其他函数。在每个回调函数中, 可以通过 `snd_pcm_substream` 的 `number` 成员得知目前操作的究竟是哪个子流, 如下所示:

```
struct snd_pcm_substream *substream;
int index = substream->number;
```

一种习惯的做法是在驱动中定义一个 PCM “构造函数”, 负责 PCM 实例的创建, 如代码清单 17.7 所示。

代码清单 17.7 PCM 设备的“构造函数”

```
1 static int __devinit snd_xxxchip_new_pcm(struct xxxchip *chip)
2 {
3     struct snd_pcm *pcm;
4     int err;
5     /* 创建 PCM 实例 */
6     if ((err = snd_pcm_new(chip->card, "xxx Chip", 0, 1, 1, &pcm)) < 0)
7         return err;
8     pcm->private_data = chip; /* 置 pcm->private_data 为芯片特定数据*/
9     strcpy(pcm->name, "xxx Chip");
10    chip->pcm = pcm;
11    ...
12    return 0;
13 }
```

2. 设置 PCM 操作

```
void snd_pcm_set_ops(struct snd_pcm *pcm, int direction, struct snd_pcm_ops *ops);
```

第 1 个参数是 `snd_pcm` 的指针，第 2 个参数是 `SNDRV_PCM_STREAM_PLAYBACK` 或 `SNDRV_PCM_STREAM_CAPTURE`，而第 3 个参数是 PCM 操作结构体 `snd_pcm_ops`，这个结构体的定义如代码清单 17.8 所示。

代码清单 17.8 `snd_pcm_ops` 结构体

```

1 struct snd_pcm_ops {
2     int (*open)(struct snd_pcm_substream *substream);
3     int (*close)(struct snd_pcm_substream *substream);
4     int (*ioctl)(struct snd_pcm_substream *substream,
5                 unsigned int cmd, void *arg);
6     int (*hw_params)(struct snd_pcm_substream *substream,
7                     struct snd_pcm_hw_params *params);
8     int (*hw_free)(struct snd_pcm_substream *substream); /* 资源释放*/
9     int (*prepare)(struct snd_pcm_substream *substream);
10    /* 在 PCM 被开始、停止或暂停时调用*/
11    int (*trigger)(struct snd_pcm_substream *substream, int cmd);
12    snd_pcm_uframes_t (*pointer)(struct snd_pcm_substream *substream); /* 当前缓冲区的
硬件位置*/
13    /* 缓冲区复制*/
14    int (*copy)(struct snd_pcm_substream *substream, int channel,
15               snd_pcm_uframes_t pos,
16               void __user *buf, snd_pcm_uframes_t count);
17    int (*silence)(struct snd_pcm_substream *substream, int channel,
18                  snd_pcm_uframes_t pos, snd_pcm_uframes_t count);
19    struct page * (*page)(struct snd_pcm_substream *substream,
20                           unsigned long offset);
21    int (*mmap)(struct snd_pcm_substream *substream, struct vm_area_struct *vma);
22    int (*ack)(struct snd_pcm_substream *substream);
23 };

```

`snd_pcm_ops` 中的所有操作都需事先通过 `snd_pcm_substream_chip()` 获得 `xxxchip` 指针，例如：

```

int xxx()
{
    struct xxxchip *chip = snd_pcm_substream_chip(substream);
    ...
}

```

当一个 PCM 子流被打开时，`snd_pcm_ops` 中的 `open()` 函数将被调用，在这个函数中，至少需要初始化 `runtime->hw` 字段，代码清单 17.9 所示为 `open()` 函数的范例。

代码清单 17.9 `snd_pcm_ops` 结构体中的 `open()` 函数

```

1 static int snd_xxx_open(struct snd_pcm_substream *substream)
2 {
3     /* 从子流获得 xxxchip 指针*/
4     struct xxxchip *chip = snd_pcm_substream_chip(substream);
5     /* 获得 PCM 运行时信息指针*/
6     struct snd_pcm_runtime *runtime = substream->runtime;
7     ...
8     /* 初始化 runtime->hw */
9     runtime->hw = snd_xxxchip_playback_hw;
10    return 0;
11 }

```



上述代码中的 `snd_xxxchip_playback_hw` 是预先定义的硬件描述。在 `open()` 函数中, 可以分配一段私有数据。如果硬件配置需要更多的限制, 也需设置硬件限制。

当 PCM 子流被关闭时, `close()` 函数将被调用。如果 `open()` 函数中分配了私有数据, 则在 `close()` 函数中应该释放 `substream` 的私有数据, 代码清单 17.10 所示为 `close()` 函数的范例。

代码清单 17.10 `snd_pcm_ops` 结构体中的 `close()` 函数

```
1 static int snd_xxx_close(struct snd_pcm_substream *substream)
2 {
3     /* 释放子流私有数据*/
4     kfree(substream->runtime->private_data);
5     ...
6 }
```

驱动中通常可以给 `snd_pcm_ops` 的 `ioctl()` 成员函数传递通用的 `snd_pcm_lib_ioctl()` 函数。

`snd_pcm_ops` 的 `hw_params()` 成员函数将在应用程序设置硬件参数 (PCM 子流的周期大小、缓冲区大小和格式等) 的时候被调用, 它的形式如下:

```
static int snd_xxx_hw_params(struct snd_pcm_substream *substream, struct snd_pcm_hw_params
*hw_params);
```

在这个函数中, 将完成大量硬件设置, 甚至包括缓冲区分配, 这时可调用如下辅助函数:

```
snd_pcm_lib_malloc_pages(substream, params_buffer_bytes(hw_params));
```

仅当 DMA 缓冲区已被预先分配的情况下, 上述调用才可成立。

与 `hw_params()` 对应的函数是 `hw_free()`, 它释放由 `hw_params()` 分配的资源, 例如, 通过如下调用释放 `snd_pcm_lib_malloc_pages()` 缓冲区:

```
snd_pcm_lib_free_pages(substream);
```

当 PCM 被“准备”时, `prepare()` 函数将被调用, 在其中可以设置采样率、格式等。`prepare()` 函数与 `hw_params()` 函数的不同在于对 `prepare()` 的调用发生在 `snd_pcm_prepare()` 每次被调用的时候。`prepare()` 的形式如下:

```
static int snd_xxx_prepare(struct snd_pcm_substream *substream);
```

`trigger()` 成员函数在 PCM 被开始、停止或暂停时调用, 函数的形式如下:

```
static int snd_xxx_trigger(struct snd_pcm_substream *substream, int cmd);
```

`cmd` 参数定义了具体的行为, 在 `trigger()` 成员函数中至少要处理 `SNDRV_PCM_TRIGGER_START` 和 `SNDRV_PCM_TRIGGER_STOP` 命令, 如果 PCM 支持暂停, 还应处理 `SNDRV_PCM_TRIGGER_PAUSE_PUSH` 和 `SNDRV_PCM_TRIGGER_PAUSE_RELEASE` 命令。如果设备支持挂起/恢复, 当能量管理状态发生变化时将处理 `SNDRV_PCM_TRIGGER_SUSPEND` 和 `SNDRV_PCM_TRIGGER_RESUME` 这两个命令。注意 `trigger()` 函数是原子的, 中途不能睡眠。代码清单 17.11 所示为 `trigger()` 函数的范例。

代码清单 17.11 `snd_pcm_ops` 结构体中的 `trigger()` 函数

```
1 static int snd_xxx_trigger(struct snd_pcm_substream *substream, int cmd)
2 {
3     switch (cmd) {
4         case SNDRV_PCM_TRIGGER_START:
5             /* 开启 PCM 引擎*/
6             break;
7         case SNDRV_PCM_TRIGGER_STOP:
8             /* 停止 PCM 引擎*/
```

```

9      break;
10     ... /* 其他命令*/
11     default:
12         return - EINVAL;
13     }
14 }

```

pointer()函数用于 PCM 中间层查询目前缓冲区的硬件位置，该函数以帧的形式返回 0~buffer_size - 1 的位置（ALSA 0.5.x 中为字节形式），此函数也是原子的。

copy()和 silence()函数一般可以省略，但是，当硬件缓冲区不处于常规内存中时需要。例如，一些设备有自己的不能被映射的硬件缓冲区，这种情况下，我们不得不将数据从内存缓冲区复制到硬件缓冲区。当内存缓冲区在物理和虚拟地址上都不连续时，这两个函数也必须被实现。

3. 分配缓冲区

分配缓冲区的最简单方法是调用如下函数：

```

int snd_pcm_lib_preallocate_pages_for_all(struct snd_pcm *pcm,
int type, void *data, size_t size, size_t max);

```

type 参数是缓冲区的类型，包含 SNDRV_DMA_TYPE_UNKNOWN（未知）、SNDRV_DMA_TYPE_CONTINUOUS（连续的非 DMA 内存）、SNDRV_DMA_TYPE_DEV（连续的通用设备）、SNDRV_DMA_TYPE_DEV_SG（通用设备 SG-buffer）和 SNDRV_DMA_TYPE_SBUS（连续的 SBUS）。如下代码将分配 64KB 的缓冲区：

```

snd_pcm_lib_preallocate_pages_for_all(pcm, SNDRV_DMA_TYPE_DEV,
snd_dma_pci_data(chip->pci), 64*1024, 64*1024);

```

4. 设置标志

在构造 PCM 实例、设置操作集并分配缓冲区之后，如果有需要，应设置 PCM 的信息标志，例如，如果 PCM 设备只支持半双工，则这样定义标志：

```

pcm->info_flags = SNDRV_PCM_INFO_HALF_DUPLEX;

```

5. PCM 实例析构

PCM 实例的“析构函数”并非是必须的，因为 PCM 实例会被 PCM 中间层代码自动释放，如果驱动中分配了一些特别的内存空间，则必须定义“析构函数”，代码清单 17.12 所示为 PCM “析构函数”与对应的“构造函数”，“析构函数”会释放“构造函数”中创建的 xxx_private_pcm_data。

代码清单 17.12 PCM 设备“析构函数”

```

1 static void xxxchip_pcm_free(struct snd_pcm *pcm)
2 {
3     /* 从 pcm 实例得到 chip */
4     struct xxxchip *chip = snd_pcm_chip(pcm);
5     /* 释放自定义用途的内存 */
6     kfree(chip->xxx_private_pcm_data);
7     ...
8 }
9
10 static int __devinit snd_xxxchip_new_pcm(struct xxxchip *chip)
11 {
12     struct snd_pcm *pcm;
13     ...
14     /* 分配自定义用途的内存 */
15     chip->xxx_private_pcm_data = kmalloc(...);

```



```
16     pcm->private_data = chip;
17     /* 设置“析构函数” */
18     pcm->private_free = xxxchip_pcm_free;
19     ...
20 }
```

上述代码第4行的 `snd_pcm_chip()` 从 PCM 实例指针获得 `xxxchip` 指针, 实际上它就是返回第16行给 PCM 实例赋予的 `xxxchip` 指针。

6. PCM 信息运行时结构体

当 PCM 子流被打开后, PCM 运行时实例(定义为结构体 `snd_pcm_runtime`, 如代码清单 17.13 所示)将被分配给这个子流, 这个指针通过 `substream->runtime` 获得。运行时指针包含各种各样的信息: `hw_params` 及 `sw_params` 配置的拷贝、缓冲区指针、mmap 记录、自旋锁等, 几乎 PCM 的所有控制信息均能从中取得。

代码清单 17.13 `snd_pcm_runtime` 结构体

```
1 struct snd_pcm_runtime {
2     /* 状态 */
3     struct snd_pcm_substream *trigger_master;
4     snd_timestamp_t trigger_tstamp; /* 触发时间戳 */
5     int overrange;
6     snd_pcm_uframes_t avail_max;
7     snd_pcm_uframes_t hw_ptr_base; /* 缓冲区复位时的位置 */
8     snd_pcm_uframes_t hw_ptr_interrupt; /* 中断时的位置 */
9     /* 硬件参数 */
10    snd_pcm_access_t access; /* 存取模式 */
11    snd_pcm_format_t format; /* SNDRV_PCM_FORMAT_* */
12    snd_pcm_subformat_t subformat; /* 子格式 */
13    unsigned int rate; /* rate in Hz */
14    unsigned int channels; /* 通道 */
15    snd_pcm_uframes_t period_size; /* 周期大小 */
16    unsigned int periods; /* 周期数 */
17    snd_pcm_uframes_t buffer_size; /* 缓冲区大小 */
18    unsigned int tick_time; /* tick time */
19    snd_pcm_uframes_t min_align; /* 格式对应的最小对齐 */
20    size_t byte_align;
21    unsigned int frame_bits;
22    unsigned int sample_bits;
23    unsigned int info;
24    unsigned int rate_num;
25    unsigned int rate_den;
26    /* 软件参数 */
27    struct timespec tstamp_mode; /* mmap 时间戳被更新 */
28    unsigned int period_step;
29    unsigned int sleep_min; /* 睡眠的最小节拍 */
30    snd_pcm_uframes_t xfer_align;
31    snd_pcm_uframes_t start_threshold;
32    snd_pcm_uframes_t stop_threshold;
33    snd_pcm_uframes_t silence_threshold; /* Silence 填充阈值 */
34    snd_pcm_uframes_t silence_size; /* Silence 填充大小 */
35    snd_pcm_uframes_t boundary;
36    snd_pcm_uframes_t silenced_start;
37    snd_pcm_uframes_t silenced_size;
```



```

38  snd_pcm_sync_id_t sync; /* 硬件同步 ID */
39  /* mmap */
40  volatile struct snd_pcm_mmap_status *status;
41  volatile struct snd_pcm_mmap_control *control;
42  atomic_t mmap_count;
43  /* 锁/调度 */
44  spinlock_t lock;
45  wait_queue_head_t sleep;
46  struct timer_list tick_timer;
47  struct fasync_struct *fasync;
48  /* 私有段 */
49  void *private_data;
50  void(*private_free)(struct snd_pcm_runtime *runtime);
51  /* 硬件描述 */
52  struct snd_pcm_hardware hw;
53  struct snd_pcm_hw_constraints hw_constraints;
54  /* 中断回调函数 */
55  void(*transfer_ack_begin)(struct snd_pcm_substream *substream);
56  void(*transfer_ack_end)(struct snd_pcm_substream *substream);
57  /* 定时器 */
58  unsigned int timer_resolution; /* timer resolution */
59  /* DMA */
60  unsigned char *dma_area; /* DMA 区域*/
61  dma_addr_t dma_addr; /* 总线物理地址*/
62  size_t dma_bytes; /* DMA 区域大小 */
63  struct snd_dma_buffer *dma_buffer_p; /* 被分配的缓冲区 */
64  #if defined(CONFIG_SND_PCM_OSS) || defined(CONFIG_SND_PCM_OSS_MODULE)
65  /* OSS 信息 */
66  struct snd_pcm_oss_runtime oss;
67  #endif
68 };

```

`snd_pcm_runtime` 中的大多数记录对被声卡驱动操作集中的函数是只读的，仅仅 PCM 中间层可从更新或修改这些信息，但是硬件描述、中断回调函数、DMA 缓冲区信息和私有数据是例外的。

下面解释 `snd_pcm_runtime` 结构体中的几个重要成员。

(1) 硬件描述。

硬件描述 (`snd_pcm_hardware` 结构体) 包含了基本硬件配置的定义，需要在 `open()` 函数中赋值。`runtime` 实例保存的是硬件描述的拷贝而非指针，这意味着在 `open()` 函数中可以修改被拷贝的描述 (`runtime->hw`)，例如：

```

struct snd_pcm_runtime *runtime = substream->runtime;
...
runtime->hw = snd_xxchip_playback_hw; /* generic 的硬件描述 */
/* 特定的硬件描述 */
if (chip->model == VERY_OLD_ONE)
    runtime->hw.channels_max = 1;

```

`snd_pcm_hardware` 结构体的定义如代码清单 17.14 所示。

代码清单 17.14 `snd_pcm_hardware` 结构体

```

1 struct snd_pcm_hardware {
2     unsigned int info; /* SNDRV_PCM_INFO_* /

```



```
3  u64 formats; /* SNDRV_PCM_FMTBIT_* */
4  unsigned int rates; /* SNDRV_PCM_RATE_* */
5  unsigned int rate_min; /* 最小采样率 */
6  unsigned int rate_max; /* 最大采样率 */
7  unsigned int channels_min; /* 最小的通道数 */
8  unsigned int channels_max; /* 最大的通道数 */
9  size_t buffer_bytes_max; /* 最大缓冲区大小 */
10 size_t period_bytes_min; /* 最小周期大小 */
11 size_t period_bytes_max; /* 最大周期大小 */
12 unsigned int periods_min; /* 最小周期数 */
13 unsigned int periods_max; /* 最大周期数 */
14 size_t fifo_size; /* FIFO 字节数 */
15 };
```

`snd_pcm_hw` 结构体中的 `info` 字段标识 PCM 设备的类型和能力, 形式为 `SNDRV_PCM_INFO_XXX`。`info` 字段至少需要定义是否支持 `mmap`, 当支持时, 应设置 `SNDRV_PCM_INFO_MMAP` 标志; 当硬件支持 `interleaved` 或 `non-interleaved` 格式时, 应设置 `SNDRV_PCM_INFO_INTERLEAVED` 或 `SNDRV_PCM_INFO_NONINTERLEAVED` 标志; 如果都支持, 则两者都可设置。

`MMAP_VALID` 和 `BLOCK_TRANSFER` 标志针对 OSS `mmap`, 只有 `mmap` 被真正支持时, 才可设置 `MMAP_VALID`; `SNDRV_PCM_INFO_PAUSE` 意味着设备可支持暂停操作, 而 `SNDRV_PCM_INFO_RESUME` 意味着设备可支持挂起/恢复操作; 当 PCM 子流能被同步, 如同步播放和录音流的 `start/stop`, 可设置 `SNDRV_PCM_INFO_SYNC_START` 标志。

`formats` 包含 PCM 设备支持的格式, 形式为 `SNDRV_PCM_FMTBIT_XXX`, 如果设备支持多种模式, 应将各种模式标志进行“或”操作。

`rates` 包含了 PCM 设备支持的采样率, 形式如 `SNDRV_PCM_RATE_XXX`, 如果支持连续的采样率, 则传递 `CONTINUOUS`。

`rate_min` 和 `rate_max` 分别定义了最大和最小的采样率, 注意: 要与 `rates` 字段相符。

`channel_min` 和 `channel_max` 定义了最大和最小的通道数量。

`buffer_bytes_max` 定义最大的缓冲区大小, 注意: 没有 `buffer_bytes_min` 字段, 这是因为它可以通过最小的周期大小和最小的周期数量计算出来。

`period` 信息与 OSS 中的 `fragment` 对应, 定义了 PCM 中断产生的周期。更小的周期大小意味着更多的中断, 在录音时, 周期大小定义了输入延迟, 在播放时, 整个缓冲区大小对应着输出延迟。

PCM 可被应用程序通过 `alsa-lib` 发送 `hw_params` 来配置, 配置信息将保存在运行时实例中。对缓冲区和周期大小的配置以帧形式存储, 而 `frames_to_bytes()` 和 `bytes_to_frames()` 可完成帧和字节的转换, 如:

```
period_bytes = frames_to_bytes(runtime, runtime->period_size);
```

(2) DMA 缓冲区信息。

包含 `dma_area` (逻辑地址)、`dma_addr` (物理地址)、`dma_bytes` (缓冲区大小) 和 `dma_private` (被 ALSA DMA 分配器使用)。可以由 `snd_pcm_lib_malloc_pages()` 实现, ALSA 中间层会设置 DMA 缓冲区信息的相关字段, 这种情况下, 驱动中不能再写这些信息, 只能读取。也就是说, 如果使用标准的缓冲区分配函数 `snd_pcm_lib_malloc_pages()` 分配缓冲区, 则我们不需要自己维护 DMA 缓冲区信息。如果缓冲区由自己分配, 则需要在 `hw_params()` 函数中管理缓冲区信息, 至少需管理 `dma_bytes` 和 `dma_addr`, 如果支持 `mmap`, 则必须管理 `dma_area`, 对 `dma_private` 的管理视情况而定。

(3) 运行状态。

通过 `runtime->status` 可以获得运行状态，它是 `snd_pcm_mmap_status` 结构体的指针，例如，通过 `runtime->status->hw_ptr` 可以获得目前的 DMA 硬件指针。此外，通过 `runtime->control` 可以获得 DMA 应用指针，它指向 `snd_pcm_mmap_control` 结构体指针，但是不建议直接访问该指针。

(4) 私有数据。

驱动中可以为子流分配一段内存并赋值给 `runtime->private_data`，注意不要与 `pcm->private_data` 混淆，后者一般指向 `xxxchip`，而前者是在 PCM 设备的 `open()` 函数中分配的动态数据，例如：

```
static int snd_xxx_open(struct snd_pcm_substream *substream)
{
    struct xxx_pcm_data *data;
    ....
    data = kmalloc(sizeof(*data), GFP_KERNEL);
    substream->runtime->private_data = data; /* 赋值 runtime->private_data */
    ....
}
```

(5) 中断回调函数：

`transfer_ack_begin()` 和 `transfer_ack_end()` 函数分别在 `snd_pcm_period_elapsed()` 的开始和结束时被调用。

根据以上分析，代码清单 17.15 给出了一个完整的 PCM 设备接口模板。

代码清单 17.15 PCM 设备接口模板

```
1  #include <sound/pcm.h>
2  ....
3  /* 播放设备硬件定义 */
4  static struct snd_pcm_hardware snd_xxxchip_playback_hw = {
5      .info = (SNDRV_PCM_INFO_MMAP | SNDRV_PCM_INFO_INTERLEAVED |
6              SNDRV_PCM_INFO_BLOCK_TRANSFER | SNDRV_PCM_INFO_MMAP_VALID),
7      .formats = SNDRV_PCM_FMTBIT_S16_LE,
8      .rates = SNDRV_PCM_RATE_8000_48000,
9      .rate_min = 8000,
10     .rate_max = 48000,
11     .channels_min = 2,
12     .channels_max = 2,
13     .buffer_bytes_max = 32768,
14     .period_bytes_min = 4096,
15     .period_bytes_max = 32768,
16     .periods_min = 1,
17     .periods_max = 1024,
18 };
19
20 /* 录音设备硬件定义 */
21 static struct snd_pcm_hardware snd_xxxchip_capture_hw = {
22     .info = (SNDRV_PCM_INFO_MMAP | SNDRV_PCM_INFO_INTERLEAVED |
23             SNDRV_PCM_INFO_BLOCK_TRANSFER | SNDRV_PCM_INFO_MMAP_VALID),
24     .formats = SNDRV_PCM_FMTBIT_S16_LE,
25     .rates = SNDRV_PCM_RATE_8000_48000,
26     .rate_min = 8000,
27     .rate_max = 48000,
28     .channels_min = 2,
29     .channels_max = 2,
```



```
30     .buffer_bytes_max = 32768,
31     .period_bytes_min = 4096,
32     .period_bytes_max = 32768,
33     .periods_min = 1,
34     .periods_max = 1024,
35 };
36
37 /* 播放: 打开函数 */
38 static int snd_xxxchip_playback_open(struct snd_pcm_substream*substream)
39 {
40     struct xxxchip *chip = snd_pcm_substream_chip(substream);
41     struct snd_pcm_runtime *runtime = substream->runtime;
42     runtime->hw = snd_xxxchip_playback_hw;
43     ... /* 硬件初始化代码*/
44     return 0;
45 }
46
47 /* 播放: 关闭函数 */
48 static int snd_xxxchip_playback_close(struct snd_pcm_substream*substream)
49 {
50     struct xxxchip *chip = snd_pcm_substream_chip(substream);
51     /* 硬件相关的代码*/
52     return 0;
53 }
54
55 /* 录音: 打开函数 */
56 static int snd_xxxchip_capture_open(struct snd_pcm_substream*substream)
57 {
58     struct xxxchip *chip = snd_pcm_substream_chip(substream);
59     struct snd_pcm_runtime *runtime = substream->runtime;
60     runtime->hw = snd_xxxchip_capture_hw;
61     ... /* 硬件初始化代码*/
62     return 0;
63 }
64
65 /* 录音: 关闭函数 */
66 static int snd_xxxchip_capture_close(struct snd_pcm_substream*substream)
67 {
68     struct xxxchip *chip = snd_pcm_substream_chip(substream);
69     ... /* 硬件相关的代码*/
70     return 0;
71 }
72 /* hw_params 函数 */
73 static int snd_xxxchip_pcm_hw_params(struct snd_pcm_substream*substream, struct
74     snd_pcm_hw_params *hw_params)
75 {
76     return snd_pcm_lib_malloc_pages(substream, params_buffer_bytes(hw_params));
77 }
78 /* hw_free 函数 */
79 static int snd_xxxchip_pcm_hw_free(struct snd_pcm_substream*substream)
80 {
81     return snd_pcm_lib_free_pages(substream);
82 }
83 /* prepare 函数 */
84 static int snd_xxxchip_pcm_prepare(struct snd_pcm_substream*substream)
```

```

85 {
86     struct xxxchip *chip = snd_pcm_substream_chip(substream);
87     struct snd_pcm_runtime *runtime = substream->runtime;
88     /* 根据目前的配置信息设置硬件
89      * 例如:
90      */
91     xxxchip_set_sample_format(chip, runtime->format);
92     xxxchip_set_sample_rate(chip, runtime->rate);
93     xxxchip_set_channels(chip, runtime->channels);
94     xxxchip_set_dma_setup(chip, runtime->dma_addr, chip->buffer_size, chip
95         ->period_size);
96     return 0;
97 }
98 /* trigger 函数 */
99 static int snd_xxxchip_pcm_trigger(struct snd_pcm_substream*substream, int cmd)
100 {
101     switch (cmd) {
102     case SNDRV_PCM_TRIGGER_START:
103         /* do something to start the PCM engine */
104         break;
105     case SNDRV_PCM_TRIGGER_STOP:
106         /* do something to stop the PCM engine */
107         break;
108     default:
109         return -EINVAL;
110     }
111 }
112
113 /* pointer 函数 */
114 static snd_pcm_uframes_t snd_xxxchip_pcm_pointer(struct snd_pcm_substream
115     *substream)
116 {
117     struct xxxchip *chip = snd_pcm_substream_chip(substream);
118     unsigned int current_ptr;
119     /*获得当前的硬件指针*/
120     current_ptr = xxxchip_get_hw_pointer(chip);
121     return current_ptr;
122 }
123 /* 放音设备操作集 */
124 static struct snd_pcm_ops snd_xxxchip_playback_ops = {
125     .open = snd_xxxchip_playback_open,
126     .close = snd_xxxchip_playback_close,
127     .ioctl = snd_pcm_lib_ioctl,
128     .hw_params = snd_xxxchip_pcm_hw_params,
129     .hw_free = snd_xxxchip_pcm_hw_free,
130     .prepare = snd_xxxchip_pcm_prepare,
131     .trigger = snd_xxxchip_pcm_trigger,
132     .pointer = snd_xxxchip_pcm_pointer,
133 };
134 /* 录音设备操作集 */
135 static struct snd_pcm_ops snd_xxxchip_capture_ops = {
136     .open = snd_xxxchip_capture_open,
137     .close = snd_xxxchip_capture_close,
138     .ioctl = snd_pcm_lib_ioctl,

```



```
139 .hw_params = snd_xxxchip_pcm_hw_params,
140 .hw_free = snd_xxxchip_pcm_hw_free,
141 .prepare = snd_xxxchip_pcm_prepare,
142 .trigger = snd_xxxchip_pcm_trigger,
143 .pointer = snd_xxxchip_pcm_pointer,
144 };
145
146 /* 创建一个 PCM 设备 */
147 static int __devinit snd_xxxchip_new_pcm(struct xxxchip *chip)
148 {
149     struct snd_pcm *pcm;
150     int err;
151     if ((err = snd_pcm_new(chip->card, "xxx Chip", 0, 1, 1, &pcm)) < 0)
152         return err;
153     pcm->private_data = chip;
154     strcpy(pcm->name, "xxx Chip");
155     chip->pcm = pcm;
156     /* 设置操作集 */
157     snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_PLAYBACK, &snd_xxxchip_playback_ops);
158     snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_CAPTURE, &snd_xxxchip_capture_ops);
159     /* 分配缓冲区 */
160     snd_pcm_lib_preallocate_pages_for_all(pcm, SNDRV_DMA_TYPE_DEV,
161         snd_dma_pci_data(chip -> pci), 64 * 1024, 64 * 1024);
162     return 0;
163 }
```

17.4.4 控制接口

1. control

控制接口对于许多开关 (switch) 和调节器 (slider) 而言应用相当广泛, 它能从用户空间被存取。control 的最主要用途是 mixer, 所有的 mixer 元素基于 control 内核 API 实现, 在 ALSA 中, control 用 `snd_kcontrol` 结构体描述。

ALSA 有一个定义很好的 AC97 控制模块, 对于仅支持 AC97 的芯片而言, 不必实现本节的内容。

创建一个新的 control 至少需要实现 `snd_kcontrol_new` 中的 `info()`、`get()` 和 `put()` 这 3 个成员函数, `snd_kcontrol_new` 结构体的定义如代码清单 17.16 所示。

代码清单 17.16 `snd_kcontrol_new` 结构体

```
1 struct snd_kcontrol_new {
2     snd_ctl_elem_iface_t iface; /* 接口 ID, SNDRV_CTL_ELEM_IFACE_XXX */
3     unsigned int device; /* 设备号 */
4     unsigned int subdevice; /* 子流 (子设备) 号 */
5     unsigned char *name; /* 名称 (ASCII 格式) */
6     unsigned int index; /* 索引 */
7     unsigned int access; /* 访问权限 */
8     unsigned int count; /* 享用元素的数量 */
9     snd_kcontrol_info_t *info;
10    snd_kcontrol_get_t *get;
11    snd_kcontrol_put_t *put;
12    unsigned long private_value;
13 };
```

iface 字段定义了 control 的类型, 形式为 SNDRV_CTL_ELEM_IFACE_XXX, 通常是 MIXER, 对于不属于 mixer 的全局控制, 使用 CARD。如果关联于某类设备, 则使用 HWDEP、PCM、RAWMIDI、TIMER 或 SEQUENCER。

name 是名称标识字符串, control 的名称非常重要, 因为 control 的作用由名称来区分。对于名称相同的 control, 则使用 index 区分。name 定义的标准是“SOURCE DIRECTION FUNCTION”即“源 方向 功能”, SOURCE 定义了 control 的源, 如“Master”、“PCM”、“CD”和“Line”, 方向则为“Playback”、“Capture”、“Bypass Playback”或“Bypass Capture”, 如果方向省略, 意味着 playback 和 capture 双向, 第 3 个参数可以是“Switch”、“Volume”和“Route”等。

“SOURCE DIRECTION FUNCTION”格式的名称例子如“Master Capture Switch”、“PCM Playback Volume”。

下面几种 control 的命名不采用“SOURCE DIRECTION FUNCTION”格式, 属于例外。

(1) 全局控制。

“Capture Source”、“Capture Switch”和“Capture Volume”用于全局录音源、输入开关和录音音量控制; “Playback Switch”、“Playback Volume”用于全局输出开关和音量控制。

(2) 音调控制。

音调控制名称的形式为“Tone Control - XXX”, 例如“Tone Control - Switch”、“Tone Control - Bas”和“Tone Control - Center”。

(3) 3D 控制。

3D 控制名称的形式为“3D Control - XXX”, 例如“3D Control - Switch”、“3D Control - Center”和“3D Control - Space”。

(4) 麦克风增益 (Mic boost)。

麦克风增益被设置为“Mic Boost”或“Mic Boost (6dB)”。

snd_kcontrol_new 结构体的 access 字段是访问控制权限, 形式如 SNDRV_CTL_ELEM_ACCESS_XXX。SNDRV_CTL_ELEM_ACCESS_READ 意味着只读, 这时 put() 函数不必实现; SNDRV_CTL_ELEM_ACCESS_WRITE 意味着只写, 这时 get() 函数不必实现。若 control 值频繁变化, 则需定义 VOLATILE 标志。当 control 处于非激活状态时, 应设置 INACTIVE 标志。

private_value 字段包含一个长整型值, 可以通过它给 info()、get() 和 put() 函数传递参数。

2. info() 函数

snd_kcontrol_new 结构体中的 info() 函数用于获得该 control 的详细信息, 该函数必须填充传递给它的第二个参数 snd_ctl_elem_info 结构体, info() 函数的形式如下:

```
static int snd_xxctl_info(struct snd_kcontrol *kcontrol, struct snd_ctl_elem_info *uinfo);
```

snd_ctl_elem_info 结构体的定义如代码清单 17.17 所示。

代码清单 17.17 snd_ctl_elem_info 结构体

```
1 struct snd_ctl_elem_info
2 {
3     struct snd_ctl_elem_id id; /* W: 元素 ID */
4     snd_ctl_elem_type_t type; /* R: 值类型 - SNDRV_CTL_ELEM_TYPE_* */
5     unsigned int access; /* R: 值访问权限 (位掩码) - SNDRV_CTL_ELEM_ACCESS_* */
```



```
6 unsigned int count; /* 值的计数 */
7 pid_t owner; /* 该 control 的拥有者 PID */
8 union {
9     struct {
10         long min; /* R: 最小值 */
11         long max; /* R: 最大值 */
12         long step; /* R: 值步进 (0 可变的) */
13     } integer;
14     struct {
15         long long min; /* R: 最小值 */
16         long long max; /* R: 最大值 */
17         long long step; /* R: 值步进 (0 可变的) */
18     } integer64;
19     struct {
20         unsigned int items; /* R: 项目数 */
21         unsigned int item; /* W: 项目号 */
22         char name[64]; /* R: 值名称 */
23     } enumerated; /* 枚举 */
24     unsigned char reserved[128];
25 }
26 value;
27 union {
28     unsigned short d[4];
29     unsigned short *d_ptr;
30 } dimen;
31 unsigned char reserved[64-4 * sizeof(unsigned short)];
32 };
```

snd_ctl_elem_info 结构体的 type 字段定义了 control 的类型,包括 BOOLEAN、INTEGER、ENUMERATED、BYTES、IEC958 和 INTEGER64。count 字段定义了这个 control 中包含的元素的数量,例如一个立体声音量 control 的 count = 2。value 是一个联合体,其所存储的值的具体类型依赖于 type。代码清单 17.18 所示为一个 info()函数填充 snd_ctl_elem_info 结构体的范例。

代码清单 17.18 snd_ctl_elem_info 结构体中的 info()函数范例

```
1 static int snd_xxxctl_info(struct snd_kcontrol *kcontrol, struct
2     snd_ctl_elem_info *uinfo)
3 {
4     uinfo->type = SNDRV_CTL_ELEM_TYPE_BOOLEAN; /* 类型为 BOOLEAN */
5     uinfo->count = 1; /* 数量为 1 */
6     uinfo->value.integer.min = 0; /* 最小值为 0 */
7     uinfo->value.integer.max = 1; /* 最大值为 1 */
8     return 0;
9 }
```

枚举类型和其他类型略有不同,对枚举类型,应为目前项目索引设置名称字符串,如代码清单 17.19 所示。

代码清单 17.19 填充 snd_ctl_elem_info 结构体中的枚举类型值

```
1 static int snd_xxxctl_info(struct snd_kcontrol *kcontrol, struct
2     snd_ctl_elem_info *uinfo)
3 {
4     /* 值名称字符串*/
```



```

5     static char *texts[4] = {
6         "First", "Second", "Third", "Fourth"
7     };
8     uinfo->type = SNDRV_CTL_ELEM_TYPE_ENUMERATED; /* 枚举类型 */
9     uinfo->count = 1; /* 数量为 1 */
10    uinfo->value.enumerated.items = 4; /* 项目数量为 1 */
11    /* 超过 3 的项目号改为 3 */
12    if (uinfo->value.enumerated.item > 3)
13        uinfo->value.enumerated.item = 3;
14    /* 为目前项目索引复制名称字符串 */
15    strcpy(uinfo->value.enumerated.name, texts[uinfo->value.enumerated.item]);
16    return 0;
17 }

```

3. get()函数

get()函数用于得到 control 的目前值并返回用户空间，代码清单 17.20 所示为 get()函数的范例。

代码清单 17.20 snd_ctl_elem_info 结构体中的 get()函数范例

```

1 static int snd_xxxctl_get(struct snd_kcontrol *kcontrol, struct
2     snd_ctl_elem_value *ucontrol)
3 {
4     /* 从 snd_kcontrol 获得 xxxchip 指针 */
5     struct xxxchip *chip = snd_kcontrol_chip(kcontrol);
6     /* 从 xxxchip 获得值并写入 snd_ctl_elem_value */
7     ucontrol->value.integer.value[0] = get_some_value(chip);
8     return 0;
9 }

```

get()函数的第二个参数的类型为 snd_ctl_elem_value，其定义如代码清单 10.21 所示。snd_ctl_elem_value 结构体的内部也包含一个由 integer、integer64、enumerated 等组成的值联合体，它的具体类型依赖于 control 的类型和 info()函数。

代码清单 17.21 snd_ctl_elem_value 结构体

```

1 struct snd_ctl_elem_value
2 {
3     struct snd_ctl_elem_id id; /* W: 元素 ID */
4     unsigned int indirect: 1; /* W: 使用间接指针 (xxx_ptr 成员) */
5     /* 值联合体 */
6     union {
7         union {
8             long value[128];
9             long *value_ptr;
10        } integer;
11        union {
12            long long value[64];
13            long long *value_ptr;
14        } integer64;
15        union {
16            unsigned int item[128];
17            unsigned int *item_ptr;
18        } enumerated;
19        union {
20            unsigned char data[512];
21            unsigned char *data_ptr;
22        } bytes;

```



```
23     struct snd_aes_iec958 iec958;
24 }
25 value; /* 只读 */
26 struct timespec tstamp;
27 unsigned char reserved[128-sizeof(struct timespec)];
28 };
```

4. put()函数

put()用于从用户空间写入值,如果值被改变,该函数返回 1,否则返回 0;如果发生错误,该函数返回一个错误码。代码清单 17.22 所示为一个 put()函数的范例。

代码清单 17.22 snd_ctl_elem_info 结构体中的 put()函数范例

```
1 static int snd_xxxctl_put(struct snd_kcontrol *kcontrol, struct
2     snd_ctl_elem_value *ucontrol)
3 {
4     /* 从 snd_kcontrol 获得 xxxchip 指针*/
5     struct xxxchip *chip = snd_kcontrol_chip(kcontrol);
6     int changed = 0; /* 默认返回值为 0 */
7     /* 值被改变*/
8     if (chip->current_value != ucontrol->value.integer.value[0]) {
9         change_current_value(chip, ucontrol->value.integer.value[0]);
10        changed = 1; /* 返回值为 1 */
11    }
12    return changed;
13 }
```

对于 get()和 put()函数而言,如果 control 有多于一个元素,即 count>1,则每个元素都需要被返回或写入。

5. 构造 control

当所有事情准备好后,我们需要创建一个 control,调用 snd_ctl_add()和 snd_ctl_new1()这两个函数来完成,这两个函数的原型为:

```
int snd_ctl_add(struct snd_card *card, struct snd_kcontrol *kcontrol);

struct snd_kcontrol *snd_ctl_new1(const struct snd_kcontrol_new *ncontrol,
    void *private_data);
```

snd_ctl_new1()函数用于创建一个 snd_kcontrol 并返回其指针,snd_ctl_add()函数用于将创建的 snd_kcontrol 添加到对应的 card 中。

6. 变更通知

如果驱动中需要在中断服务程序中改变或更新一个 control,可以调用 snd_ctl_notify()函数,此函数原型为:

```
void snd_ctl_notify(struct snd_card *card, unsigned int mask, struct snd_ctl_elem_id *id);
```

该函数的第二个参数为事件掩码(event-mask),第三个参数为该通知的 control 元素 id 指针。

例如,如下语句定义的事件掩码 SNDRV_CTL_EVENT_MASK_VALUE 意味着 control 值的改变被通知:

```
snd_ctl_notify(card, SNDRV_CTL_EVENT_MASK_VALUE, id_pointer);
```

17.4.5 AC97 API 接口

ALSA AC97 编解码层被很好地定义,利用它,驱动工程师只需编写少量底层的控制函数。

1. AC97 实例构造

为了创建一个 AC97 实例，首先需要调用 `snd_ac97_bus()` 函数构建 AC97 总线及其操作，这个函数的原型为：

```
int snd_ac97_bus(struct snd_card *card, int num, struct snd_ac97_bus_ops *ops,
                void *private_data, struct snd_ac97_bus **rbus);
```

该函数的第 3 个参数 `ops` 是一个 `snd_ac97_bus_ops` 结构体，其定义如代码清单 17.23 所示。

代码清单 17.23 `snd_ac97_bus_ops` 结构体

```
1 struct snd_ac97_bus_ops {
2     void(*reset)(struct snd_ac97 *ac97); /* 复位函数*/
3     /* 写入函数*/
4     void(*write)(struct snd_ac97 *ac97, unsigned short reg, unsigned short val);
5     /* 读取函数*/
6     unsigned short(*read)(struct snd_ac97 *ac97, unsigned short reg);
7     void(*wait)(struct snd_ac97 *ac97);
8     void(*init)(struct snd_ac97 *ac97);
9 };
```

接下来，调用 `snd_ac97_mixer()` 函数注册混音器，这个函数的原型为：

```
int snd_ac97_mixer(struct snd_ac97_bus *bus, struct snd_ac97_template *template, struct
snd_ac97 **rac97);
```

代码清单 17.24 所示为 AC97 实例的创建过程。

代码清单 17.24 AC97 实例的创建过程范例

```
1 struct snd_ac97_bus *bus;
2 /* AC97 总线操作*/
3 static struct snd_ac97_bus_ops ops = {
4     .write = snd_mychip_ac97_write,
5     .read = snd_mychip_ac97_read,
6 };
7 /* AC97 总线与操作创建*/
8 snd_ac97_bus(card, 0, &ops, NULL, &bus);
9 /* AC97 模板*/
10 struct snd_ac97_template ac97;
11 int err;
12 memset(&ac97, 0, sizeof(ac97));
13 ac97.private_data = chip; /* 私有数据*/
14 /* 注册混音器*/
15 snd_ac97_mixer(bus, &ac97, &chip->ac97);
```

上述代码第一行的 `snd_ac97_bus` 结构体指针 `bus` 的指针被传入第 8 行的 `snd_ac97_bus()` 函数并被赋值，`chip->ac97` 的指针被传入第 15 行的 `snd_ac97_mixer()` 并被赋值，`chip->ac97` 将成员新创建 AC97 实例的指针。

如果一个声卡上包含多个编解码器，这种情况下，需要多次调用 `snd_ac97_mixer()` 并对 `snd_ac97` 的 `num` 成员（编解码器序号）赋予相应的序号。驱动中可以为不同的编解码器编写不同的 `snd_ac97_bus_ops` 成员函数中，或者只是在相同的一套成员函数中通过 `ac97.num` 获得序号后再区分进行具体的操作。

2. `snd_ac97_bus_ops` 成员函数

`snd_ac97_bus_ops` 结构体中的 `read()` 和 `write()` 成员函数完成底层的硬件访问，`reset()` 函数用于复位编解码器，`wait()` 函数用于编解码器标准初始化过程中的特定等待，如果芯片要求额



外的等待时间, 则应实现这个函数, `init()` 用于完成编解码器附加的初始化。代码清单 17.25 所示为 `read()` 和 `write()` 函数的范例。

代码清单 17.25 `snd_ac97_bus_ops` 结构体中的 `read()` 和 `write()` 函数范例

```
1 static unsigned short snd_xxxchip_ac97_read(struct snd_ac97 *ac97, unsigned
2     short reg)
3 {
4     struct xxxchip *chip = ac97->private_data;
5     ...
6     return the_register_value; /* 返回寄存器值*/
7 }
8
9 static void snd_xxxchip_ac97_write(struct snd_ac97 *ac97, unsigned short reg,
10     unsigned short val)
11 {
12     struct xxxchip *chip = ac97->private_data;
13     ...
14     /* 将被给的寄存器值写入 codec */
15 }
```

3. 修改寄存器

如果需要在驱动中访问编解码器, 可使用如下函数:

```
void snd_ac97_write(struct snd_ac97 *ac97, unsigned short reg, unsigned short value);

int snd_ac97_update(struct snd_ac97 *ac97, unsigned short reg, unsigned short value);

int snd_ac97_update_bits(struct snd_ac97 *ac97, unsigned short reg, unsigned short mask,
    unsigned short value);

unsigned short snd_ac97_read(struct snd_ac97 *ac97, unsigned short reg);
```

`snd_ac97_update()` 与 `void snd_ac97_write()` 的区别在于前者在值已经设置的情况下不会再次设置, 而后者则会再写一次。`snd_ac97_update_bits()` 用于更新寄存器的某些位, 由 `mask` 决定。

除此之外, 还有一个函数可用于设置采样率:

```
int snd_ac97_set_rate(struct snd_ac97 *ac97, int reg, unsigned int rate);
```

这个函数的第二个参数 `reg` 可以是 `AC97_PCM_MIC_ADC_RATE`、`AC97_PCM_FRONT_DAC_RATE`、`AC97_PCM_LR_ADC_RATE` 和 `AC97_SPDIF`, 对于 `AC97_SPDIF` 而言, 寄存器并非真地被改变了, 只是相应的 IEC958 状态位将被更新。

4. 时钟调整

在一些芯片上, 编解码器的时钟频率不是 48000Hz, 而是使用 PCI 时钟以节省一个晶振, 在这种情况下, 我们应该改变 `bus->clock` 为相应的值, 例如 `intel8x0` 和 `es1968` 包含时钟的自动测量函数。

5. proc 文件

ALSA AC97 接口会创建如 `/proc/asound/card0/codec97#0/ac97#0-0` 和 `ac97#0-0+regs` 这样的 proc 文件, 通过这些文件可以查看编解码器目前的状态和寄存器。

如果一个芯片上有多个 codecs, 可多次调用 `snd_ac97_mixer()`。

17.4.6 ALSA 用户空间编程

ALSA 驱动的声卡在用户空间不宜直接使用文件接口, 而应使用 `alsa-lib`, 代码清单 17.26 所

示为基于 ALSA 音频驱动的最简单的播放应用程序。

代码清单 17.26 ALSA 用户空间播放程序

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <alsa/asoundlib.h>
4
5 main(int argc, char *argv[])
6 {
7     int i;
8     int err;
9     short buf[128];
10    snd_pcm_t *playback_handle; /* PCM 设备句柄*/
11    snd_pcm_hw_params_t *hw_params; /* 硬件信息和 PCM 流配置*/
12    /* 打开 PCM, 最后一个参数为 0 意味着标准配置*/
13    if ((err = snd_pcm_open(&playback_handle, argv[1], SND_PCM_STREAM_PLAYBACK, 0)
14        ) < 0) {
15        fprintf(stderr, "cannot open audio device %s (%s)\n", argv[1], snd_strerror
16            (err));
17        exit(1);
18    }
19    /* 分配 snd_pcm_hw_params_t 结构体*/
20    if ((err = snd_pcm_hw_params_malloc(&hw_params)) < 0) {
21        fprintf(stderr, "cannot allocate hardware parameter structure (%s)\n",
22            snd_strerror(err));
23        exit(1);
24    }
25    /* 初始化 hw_params */
26    if ((err = snd_pcm_hw_params_any(playback_handle, hw_params)) < 0) {
27        fprintf(stderr, "cannot initialize hardware parameter structure (%s)\n",
28            snd_strerror(err));
29        exit(1);
30    }
31    /* 初始化访问权限*/
32    if ((err = snd_pcm_hw_params_set_access(playback_handle, hw_params,
33        SND_PCM_ACCESS_RW_INTERLEAVED)) < 0) {
34        fprintf(stderr, "cannot set access type (%s)\n", snd_strerror(err));
35        exit(1);
36    }
37    /* 初始化采样格式*/
38    if ((err = snd_pcm_hw_params_set_format(playback_handle, hw_params,
39        SND_PCM_FORMAT_S16_LE)) < 0) {
40        fprintf(stderr, "cannot set sample format (%s)\n", snd_strerror(err));
41        exit(1);
42    }
43    /* 设置采样率, 如果硬件不支持我们设置的采样率, 将使用最接近的*/
44    if ((err = snd_pcm_hw_params_set_rate_near(playback_handle, hw_params, 44100,
45        0)) < 0) {
46        fprintf(stderr, "cannot set sample rate (%s)\n", snd_strerror(err));
47        exit(1);
48    }
49    /* 设置通道数量*/
50    if ((err = snd_pcm_hw_params_set_channels(playback_handle, hw_params, 2)) < 0) {
51        fprintf(stderr, "cannot set channel count (%s)\n", snd_strerror(err));
```



```
52     exit(1);
53 }
54 /* 设置 hw_params */
55 if ((err = snd_pcm_hw_params(playback_handle, hw_params)) < 0) {
56     fprintf(stderr, "cannot set parameters (%s)\n", snd_strerror(err));
57     exit(1);
58 }
59 /* 释放分配的 snd_pcm_hw_params_t 结构体*/
60 snd_pcm_hw_params_free(hw_params);
61 /* 完成硬件参数设置, 使设备准备好*/
62 if ((err = snd_pcm_prepare(playback_handle)) < 0) {
63     fprintf(stderr, "cannot prepare audio interface for use (%s)\n",
64             snd_strerror(err));
65     exit(1);
66 }
67
68 for (i = 0; i < 10; ++i) {
69     /* 写音频数据到 PCM 设备*/
70     if ((err = snd_pcm_writei(playback_handle, buf, 128)) != 128) {
71         fprintf(stderr, "write to audio interface failed (%s)\n", snd_strerror
72                 (err));
73         exit(1);
74     }
75 }
76 /* 关闭 PCM 设备句柄*/
77 snd_pcm_close(playback_handle);
78 exit(0);
79 }
```

由上述代码可以看出, ALSA 用户空间编程的流程与 17.3.4 小节给出的 OSS 驱动用户空间编程的流程基本是一致的, 都经过了“打开—设置参数—读写音频数据”的过程, 不同在于 OSS 打开的是设备文件, 设置参数使用的是 `ioctl()` 系统调用, 读写音频数据使用的是 `read()`、`write()` 文件 API, 而 ALSA 则全部使用 `alsa-lib` 中的 API。

把上述代码第 70 行的 `snd_pcm_writei()` 函数替换为 `snd_pcm_readi()`, 变成了一个最简单的录音程序。

代码清单 17.27 的程序打开一个音频接口, 配置它为立体声、16 位、44.1kHz 采样和基于 `interleave` 的读写。它阻塞等待直接接口准备好接收放音数据, 这时候将数据复制到缓冲区。这种设计方法使得程序很容易移植到类似 JACK、LADSPA、Coreaudio、VST 等 `callback` 机制驱动的系统。

代码清单 17.27 ALSA 用户空间播放程序 (基于“中断”)

```
1  #include ...
2
3  snd_pcm_t *playback_handle;
4  short buf[4096];
5
6  int playback_callback(snd_pcm_sframes_t nframes)
7  {
8      int err;
9      printf("playback callback called with %u frames\n", nframes);
10     /* 填充缓冲区 */
11     if ((err = snd_pcm_writei(playback_handle, buf, nframes)) < 0) {
```

```

12     fprintf(stderr, "write failed (%s)\n", snd_strerror(err));
13 }
14
15     return err;
16 }
17
18 main(int argc, char *argv[])
19 {
20
21     snd_pcm_hw_params_t *hw_params;
22     snd_pcm_sw_params_t *sw_params;
23     snd_pcm_sframes_t frames_to_deliver;
24     int nfds;
25     int err;
26     struct pollfd *pfds;
27
28     if ((err = snd_pcm_open(&playback_handle, argv[1], SND_PCM_STREAM_PLAYBACK, 0)
29         ) < 0) {
30         ...
31     }
32
33     if ((err = snd_pcm_hw_params_malloc(&hw_params)) < 0) {
34         ...
35     }
36
37     if ((err = snd_pcm_hw_params_any(playback_handle, hw_params)) < 0) {
38         ...
39     }
40
41     if ((err = snd_pcm_hw_params_set_access(playback_handle, hw_params,
42         SND_PCM_ACCESS_RW_INTERLEAVED)) < 0) {
43         ...
44     }
45
46     if ((err = snd_pcm_hw_params_set_format(playback_handle, hw_params,
47         SND_PCM_FORMAT_S16_LE)) < 0) {
48         ...
49     }
50
51     if ((err = snd_pcm_hw_params_set_rate_near(playback_handle, hw_params, 44100,
52         0)) < 0) {
53         ...
54     }
55
56     if ((err = snd_pcm_hw_params_set_channels(playback_handle, hw_params, 2)) < 0) {
57         ...
58     }
59
60     if ((err = snd_pcm_hw_params(playback_handle, hw_params)) < 0) {
61         ...
62     }
63
64     snd_pcm_hw_params_free(hw_params);
65
66     /* 告诉 ALSA 当 4096 个以上帧可以传递时唤醒我们 */

```



```
67  if ((err = snd_pcm_sw_params_malloc(&sw_params)) < 0) {
68      ...
69  }
70  if ((err = snd_pcm_sw_params_current(playback_handle, sw_params)) < 0) {
71      ...
72  }
73  /* 设置 4096 帧传递一次数据 */
74  if ((err = snd_pcm_sw_params_set_avail_min(playback_handle, sw_params, 4096))
75      < 0) {
76      ...
77  }
78  /* 一旦有数据就开始播放 */
79  if ((err = snd_pcm_sw_params_set_start_threshold(playback_handle, sw_params,
80      0U)) < 0) {
81      ...
82  }
83  if ((err = snd_pcm_sw_params(playback_handle, sw_params)) < 0) {
84      ...
85  }
86
87  /* 每 4096 帧接口将中断内核, ALSA 将很快唤醒本程序 */
88
89  if ((err = snd_pcm_prepare(playback_handle)) < 0) {
90      ...
91  }
92
93  while (1) {
94      /* 等待, 直到接口准备好传递数据, 或者 1s 超时发生 */
95      if ((err = snd_pcm_wait(playback_handle, 1000)) < 0) {
96          ...
97      }
98
99      /* 查出有多少空间可放置 playback 数据 */
100     if ((frames_to_deliver = snd_pcm_avail_update(playback_handle)) < 0) {
101         if (frames_to_deliver == - EPIPE) {
102             fprintf(stderr, "an xrun occurred\n");
103             break;
104         } else {
105             fprintf(stderr, "unknown ALSA avail update return value (%d)\n",
106                 frames_to_deliver);
107             break;
108         }
109     }
110
111     frames_to_deliver = frames_to_deliver > 4096 ? 4096 : frames_to_deliver;
112
113     /* 传递数据 */
114     if (playback_callback(frames_to_deliver) != frames_to_deliver) {
115         ...
116     }
117 }
118
119 snd_pcm_close(playback_handle);
120 exit(0);
121 }
```


17.5 Linux ASoC 音频设备驱动

17.5.1 ASoC 驱动的组成

ASoC (ALSA System on Chip) 是 ALSA 在 SoC 方面的发展和演变, 它在本质上仍然属于 ALSA, 但是在 ALSA 架构基础上对 CPU 相关的代码和 Codec 相关的代码进行了分离。其原因是, 采用传统 ALSA 架构的情况下, 同一型号的 Codec 工作于不同的 CPU 时, 需要不同的驱动, 这不符合代码重用的要求。

对于目前嵌入式系统上的声卡驱动开发, 我们建议读者尽量采用 ASoC 框架, ASoC 主要由 3 部分组成。

(1) Codec 驱动。这一部分只关心 Codec 本身, 与 CPU 平台相关的特性不由此部分操作。

(2) 平台驱动。这一部分只关心 CPU 本身, 不关心 Codec。它主要处理两个问题: DMA 引擎和 SoC 集成的 PCM、I²S 或 AC '97 数字接口控制。

(3) 板驱动 (也称为 machine 驱动)。这一部分将平台驱动和 Codec 驱动绑定在一起, 描述了板一级的硬件特征。

在以上 3 部分中, 1 和 2 基本都可以仍然是通用的驱动了, 也就是说, Codec 驱动认为自己可以连接任意 CPU, 而 CPU 的 I²S、PCM 或 AC '97 接口对应的平台驱动则认为自己可以连接任意符合其接口类型的 Codec, 只有 3 是不通用的, 由特定的电路板上具体的 CPU 和 Codec 确定, 因此它很像一个插座, 上面插上了 Codec 和平台这两个插头。

在以上三部分之上的是 ASoC 核心层, 由内核源代码中的 sound/soc/soc-core.c 实现, 查看其源代码发现它完全是一个传统的 ALSA 驱动。因此, 对于基于 ASoC 架构的声卡驱动而言, alsa-lib 以及 ALSA 的一系列 utility 仍然是可用的, 如 amixer、aplay 均无需针对 ASoC 进行任何改动。而 ASoC 的用户编程方法也与 ALSA 完全一致。

内核源代码的 Documentation/sound/alsa/soc/目录包含了 ASoC 相关的文档。

17.5.2 ASoC Codec 驱动

在 ASoC 架构下, Codec 驱动负责如下工作。

(1) Codec DAI (Digital Audio Interfaces) 和 PCM 配置, 由结构体 snd_soc_dai (如代码清单 17.28) 来描述, 形容 playback、capture 的属性以及 DAI 接口的操作。

代码清单 17.28 DAI 结构体 snd_soc_dai 定义

```
1 struct snd_soc_dai {
2     /* DAI 的描述 */
3     char *name;
4     unsigned int id;
5     unsigned char type;
6
7     /* DAI callbacks */
8     int (*probe)(struct platform_device *pdev,
```



```
9         struct snd_soc_dai *dai);
10 void (*remove)(struct platform_device *pdev,
11               struct snd_soc_dai *dai);
12 int (*suspend)(struct platform_device *pdev,
13               struct snd_soc_dai *dai);
14 int (*resume)(struct platform_device *pdev,
15               struct snd_soc_dai *dai);
16
17 /* ops */
18 struct snd_soc_ops ops;
19 struct snd_soc_dai_ops dai_ops;
20
21 /* DAI 的能力 */
22 struct snd_soc_pcm_stream capture;
23 struct snd_soc_pcm_stream playback;
24
25 /* DAI 运行时信息 */
26 struct snd_pcm_runtime *runtime;
27 struct snd_soc_codec *codec;
28 unsigned int active;
29 unsigned char pop_wait:1;
30 void *dma_data;
31
32 /* DAI 私有数据 */
33 void *private_data;
34 };
```

第 22、23 行的 `snd_soc_pcm_stream` 类型成员 `capture`、`playback` 分别描述录音和放音的能力，`snd_soc_pcm_stream` 结构体主要包含 `formats`、`rates`、`rate_min`、`rate_max`、`channels_min`、`channels_max` 这几个字段。

(2) Codec IO 操作、动态音频电源管理以及时钟、PLL 等控制。

代码清单 17.28 中第 27 行的 `snd_soc_codec` 结构体是对 Codec 本身 I/O 控制以及动态音频电源管理 (Dynamic Audio Power Management, DAPM) 的描述。它描述 I²C、SPI 或 AC '97 如何读写 Codec 寄存器并容纳 DAPM 链表，其定义如代码清单 17.29，核心成员为 `read()`、`write()`、`hw_write()`、`hw_read()`、`dapm_widgets`、`dapm_paths` 等。

代码清单 17.29 `snd_soc_codec` 结构体定义

```
1 struct snd_soc_codec {
2     char *name;
3     struct module *owner;
4     struct mutex mutex;
5
6     /* callbacks */
7     int (*set_bias_level)(struct snd_soc_codec *,
8                           enum snd_soc_bias_level level);
9
10    /* runtime */
11    struct snd_card *card;
12    struct snd_ac97 *ac97; /* for ad-hoc ac97 devices */
13    unsigned int active;
14    unsigned int pcm_devs;
15    void *private_data;
```

```

16
17  /* codec IO */
18  void *control_data; /* codec control (i2c/3wire) data */
19  unsigned int (*read)(struct snd_soc_codec *, unsigned int);
20  int (*write)(struct snd_soc_codec *, unsigned int, unsigned int);
21  int (*display_register)(struct snd_soc_codec *, char *,
22                          size_t, unsigned int);
23  hw_write_t hw_write;
24  hw_read_t hw_read;
25  void *reg_cache;
26  short reg_cache_size;
27  short reg_cache_step;
28
29  /* dapm */
30  struct list_head dapm_widgets;
31  struct list_head dapm_paths;
32  enum snd_soc_bias_level bias_level;
33  enum snd_soc_bias_level suspend_bias_level;
34  struct delayed_work delayed_work;
35
36  /* codec DAI's */
37  struct snd_soc_dai *dai;
38  unsigned int num_dai;
39 };

```

代码清单 17.28 中第 19 行的 `snd_soc_dai_ops` 则描述该 Codec 的时钟、PLL 以及格式设置，主要包括 `set_sysclk()`、`set_pll()`、`set_clkdiv()`、`set_fmt()` 等成员函数，其定义如代码清单 17.30。

代码清单 17.30 `snd_soc_dai_ops` 结构体定义

```

1 struct snd_soc_dai_ops {
2     /* DAI 时钟配置 */
3     int (*set_sysclk)(struct snd_soc_dai *dai,
4                       int clk_id, unsigned int freq, int dir);
5     int (*set_pll)(struct snd_soc_dai *dai,
6                    int pll_id, unsigned int freq_in, unsigned int freq_out);
7     int (*set_clkdiv)(struct snd_soc_dai *dai, int div_id, int div);
8
9     /* DAI 格式配置 */
10    int (*set_fmt)(struct snd_soc_dai *dai, unsigned int fmt);
11    int (*set_tdm_slot)(struct snd_soc_dai *dai,
12                       unsigned int mask, int slots);
13    int (*set_tristate)(struct snd_soc_dai *dai, int tristate);
14
15    /* 数字静音 */
16    int (*digital_mute)(struct snd_soc_dai *dai, int mute);
17 };

```

(3) Codec 的 mixer 控制。

ASoC 中定义了一组宏来描述 Codec 的 mixer 控制，这组宏可以方便地将 mixer 名和对应的寄存器进行绑定，主要包括：

```

SOC_SINGLE(xname, reg, shift, mask, invert)
SOC_DOUBLE(xname, reg, shift_left, shift_right, mask, invert)
SOC_ENUM_SINGLE(xreg, xshift, xmask, xtexts)

```

例如，对于宏 `SOC_SINGLE` 而言，参数 `xname` 是 mixer 的名字（如“Playback Volume”），`reg`



是控制该 mixer 的寄存器, shift 对应寄存器内的位, mask 是进行操作时的屏蔽位, invert 表明是否倒序或翻转。

(4) Codec 音频操作。

在 ASoC 驱动的 Codec 部分, 也需要关心音频流开始采集或播放时的一些动作, 如 hw_params()、hw_free()、prepare()、trigger() 这些操作, 不过与原始 ALSA 不同的是, 在 Codec 驱动的这些函数中, 不关心 CPU 端, 而只关心 Codec 本身, 由结构体 snd_soc_ops 描述, 如代码清单 17.31 所示。

代码清单 17.31 snd_soc_ops 结构体定义

```
1 struct snd_soc_ops {
2     int (*startup)(struct snd_pcm_substream *);
3     void (*shutdown)(struct snd_pcm_substream *);
4     int (*hw_params)(struct snd_pcm_substream *, struct snd_pcm_hw_params *);
5     int (*hw_free)(struct snd_pcm_substream *);
6     int (*prepare)(struct snd_pcm_substream *);
7     int (*trigger)(struct snd_pcm_substream *, int);
8 };
```

ASoC 的主要维护者 Mark Brown (broonie@opensource.wolfsonmicro.com) 是 Wolfson 公司的成员, 因此从内核的 drivers/sound/soc/codecs 下容易发现 Wolfson 系列 Codec 芯片的驱动, 此外, Analog Devices 也是该目录源代码的主要贡献者。

17.5.3 ASoC 平台驱动

首先, 在 ASoC 平台驱动部分, 同样存在着 Codec 驱动中的 snd_soc_dai、snd_soc_dai_ops、snd_soc_ops 这 3 个结构体的实例用于描述 DAI 和 DAI 上的操作, 不过不同的是, 在平台驱动中, 它们只描述 CPU 相关的部分而不描述 Codec。除此之外, 在 ASoC 平台驱动中, 必须实现完整的 DMA 驱动, 即传统 ALSA 的 snd_pcm_ops 结构体成员函数 trigger()、pointer() 等。因此 ASoC 平台驱动通常由 DAI 和 DMA 两部分组成, 如代码清单 17.32 所示。

代码清单 17.32 ASoC 平台驱动的组成

```
1 /* DAI 部分 */
2 static int xxx_i2s_set_dai_fmt(struct snd_soc_dai *cpu_dai,
3                               unsigned int fmt)
4 {
5     ...
6 }
7
8 static int xxx_i2s_startup(struct snd_pcm_substream *substream)
9 {
10     ...
11 }
12
13 static int xxx_i2s_hw_params(struct snd_pcm_substream *substream,
14                             struct snd_pcm_hw_params *params)
15 {
16     ...
17 }
18
```

```

19 static void xxx_i2s_shutdown(struct snd_pcm_substream *substream)
20 {
21     ...
22 }
23
24 static int xxx_i2s_probe(struct platform_device *pdev,
25                          struct snd_soc_dai *dai)
26 {
27     ...
28 }
29
30 static void xxx_i2s_remove(struct platform_device *pdev,
31                            struct snd_soc_dai *dai)
32 {
33     ...
34 }
35
36 static int xxx_i2s_suspend(struct platform_device *dev,
37                            struct snd_soc_dai *dai)
38 {
39     ...
40 }
41
42 static int xxx_i2s_resume(struct platform_device *pdev,
43                           struct snd_soc_dai *dai)
44 {
45     ...
46 }
47
48 struct snd_soc_dai xxx_i2s_dai = {
49     .name = "xxx-i2s",
50     .id = 0,
51     .type = SND_SOC_DAI_I2S,
52     .probe = xxx_i2s_probe,
53     .remove = xxx_i2s_remove,
54     .suspend = xxx_i2s_suspend,
55     .resume = xxx_i2s_resume,
56     .playback = {
57         .channels_min = 1,
58         .channels_max = 2,
59         .rates = XXX_I2S_RATES,
60         .formats = XXX_I2S_FORMATS,},
61     .capture = {
62         .channels_min = 1,
63         .channels_max = 2,
64         .rates = XXX_I2S_RATES,
65         .formats = XXX_I2S_FORMATS,},
66     .ops = {
67         .startup = xxx_i2s_startup,
68         .shutdown = xxx_i2s_shutdown,
69         .hw_params = xxx_i2s_hw_params,},
70     .dai_ops = {
71         .set_fmt = xxx_i2s_set_dai_fmt,
72     },
73 };

```



```
74
75  /* DMA 部分 */
76  static void bf5xx_dma_irq(void *data)
77  {
78      struct snd_pcm_substream *pcm = data;
79      snd_pcm_period_elapsed(pcm);
80  }
81
82  static const struct snd_pcm_hw_params xxx_pcm_hw_params = {
83      ...
84  };
85
86  static int xxx_pcm_hw_params(struct snd_pcm_substream *substream,
87      struct snd_pcm_hw_params *params)
88  {
89      ...
90      snd_pcm_lib_malloc_pages(substream, size);
91
92      return 0;
93  }
94
95  static int xxx_pcm_hw_free(struct snd_pcm_substream *substream)
96  {
97      snd_pcm_lib_free_pages(substream);
98
99      return 0;
100 }
101
102 static int xxx_pcm_prepare(struct snd_pcm_substream *substream)
103 {
104     ...
105 }
106
107 static int xxx_pcm_trigger(struct snd_pcm_substream *substream, int cmd)
108 {
109     ...
110 }
111
112 static snd_pcm_uframes_t xxx_pcm_pointer(struct snd_pcm_substream *substream)
113 {
114     ...
115 }
116
117 static int xxx_pcm_open(struct snd_pcm_substream *substream)
118 {
119     ...
120 }
121
122 static int xxx_pcm_mmap(struct snd_pcm_substream *substream,
123     struct vm_area_struct *vma)
124 {
125     ...;
126 }
127
128 struct snd_pcm_ops xxx_pcm_i2s_ops = {
```

```

129     .open      = xxx_pcm_open,
130     .ioctl     = snd_pcm_lib_ioctl,
131     .hw_params  = xxx_pcm_hw_params,
132     .hw_free    = xxx_pcm_hw_free,
133     .prepare    = xxx_pcm_prepare,
134     .trigger    = xxx_pcm_trigger,
135     .pointer    = xxx_pcm_pointer,
136     .mmap       = xxx_pcm_mmap,
137 };

```

17.5.4 ASoC 板驱动

ASoC 板驱动直接与板对应，对于一块确定的电路板，其 SoC 和 Codec 都是确定的，因此板驱动将 ASoC Codec 驱动和 CPU 端的平台驱动进行绑定，这个绑定用数据结构 `snd_soc_dai_link` 描述，其定义如代码清单 17.33 所示。

代码清单 17.33 `snd_soc_dai_link` 结构体

```

1 struct snd_soc_dai_link {
2     char *name;                /* Codec name */
3     char *stream_name;        /* Stream name */
4
5     /* DAI */
6     struct snd_soc_dai *codec_dai;
7     struct snd_soc_dai *cpu_dai;
8
9     /* 板流操作 */
10    struct snd_soc_ops *ops;
11
12    /* codec/machine 特定的初始化 */
13    int (*init)(struct snd_soc_codec *codec);
14
15    /* DAI pcm */
16    struct snd_pcm *pcm;
17};

```

除此之外，板驱动还关心一些板特定的硬件操作，因此也存在一个 `snd_soc_ops` 的实例。

在板驱动的平台初始化函数中，会通过 `platform_device_add()` 注册一个名为“soc-audio”的 platform 设备，这是因为 `soc-core.c` 注册了一个名为“soc-audio”的 platform 驱动，因此，在板驱动中注册“soc-audio”设备会引起两者的匹配，从而引发一系列的初始化操作。尤其值得一提的是，“soc-audio”设备的私有数据需要为一个 `snd_soc_device` 的结构体实体，因此一个板驱动典型的模块加载函数将形如代码清单 17.34。

代码清单 17.34 ASoC 板驱动模块加载函数及其访问的数据结构

```

1 static struct snd_soc_dai_link cpux_codec_dai = {
2     .name = "codecy",
3     .stream_name = "CODECY",
4     .cpu_dai = &cpux_i2s_dai,
5     .codec_dai = &codecy_dai,
6     .ops = &cpux_codec_ops,
7 };
8

```



```
9 static struct snd_soc_machine cpux_codeccy = {
10     .name = "cpux_codeccy",
11     .probe = cpux_probe,
12     .dai_link = &cpux_codeccy_dai,
13     .num_links = 1,
14 };
15
16 static struct snd_soc_device cpux_codeccy_snd_devdata = {
17     .machine = &cpux_codeccy,
18     .platform = &cpux_i2s_soc_platform,
19     .codec_dev = &soc_codec_dev_codeccy,
20 };
21
22 static struct platform_device *cpux_codeccy_snd_device;
23
24 static int __init cpux_codeccy_init(void)
25 {
26     int ret;
27
28     cpux_codeccy_snd_device = platform_device_alloc("soc-audio", -1);
29     if (!cpux_codeccy_snd_device)
30         return -ENOMEM;
31
32     platform_set_drvdata(cpux_codeccy_snd_device, &cpux_codeccy_snd_devdata);
33     cpux_codeccy_snd_devdata.dev = &cpux_codeccy_snd_device->dev;
34     ret = platform_device_add(cpux_codeccy_snd_device);
35
36     if (ret)
37         platform_device_put(cpux_codeccy_snd_device);
38
39     return ret;
40 }
41 module_init(cpux_codeccy_init);
```

上述代码中访问的 `snd_soc_device` 是对一个 ASoC 设备的整体封装，因此其中包括了封装板用的 `snd_soc_machine` (machine 成员)、封装 ASoC Codec 设备用的 `snd_soc_codec_device` (codec_dev 成员)，封装 ASoC 平台设备用的 `snd_soc_platform` (platform 成员)。

ASoC 驱动的 Codec、平台和板驱动是 3 个独立的内核模块，在板驱动中，对 ASoC Codec 设备、ASoC 平台设备实例的访问都通过被 ASoC Codec 驱动或 ASoC 平台驱动导出的全局变量执行，这使得 ASoC 难以同时支持两个以上的 Codec。至本书截稿时，ASoC 的其中一个维护者 Liam Girdwood (lrg@slimlogic.co.uk) 正在添加 ASoC 对多 Codec 的支持。

17.6 S3C6410+WM9714 ASoC 驱动实例

LDD6410 开发板上为 S3C6410 的 AC'97 接口上连接了 Wolfson 公司的 WM9714 Codec 芯片，其硬件连接如图 17.6 所示。WM9714 芯片主要外接了 Microphone、Line in 模拟输入和 Headphone 模拟输出。而在数字接口方面，与 S3C6410 CPU 的连接包含了 AC-Link 总线上必要的信号。



图 17.7 LDD6410 开发板连接的 WM9714



WM9714 是一款比较复杂的 Codec 芯片, 其与 AC'97 2.2 兼容。内部集成了 HiFi 立体声 ADC/DAC、AUX ADC/DAC 和用于语音的 Voice ADC/DAC, 同时包含数个 mux 和 mixer 以及增益调节, 在使用该芯片上, 读懂数据手册中的“Audio Paths Overview”即音频路径图非常关键。

LDD6410 开发板上 ASoC 驱动的 3 部分分别是。

(1) Codec 驱动。由内核源代码 sound/soc/codecs/wm9713.c 实现。

(2) 平台驱动。由内核源代码 sound/soc/s3c/s3c-ac97.c 实现 S3C6410 CPU 端的 DAI 驱动, 由 sound/soc/s3c/s3c-pcm.c 实现 CPU 端的 DMA 驱动。

(3) 板驱动。由内核源代码 sound/soc/s3c/smdk6410_wm9713.c 实现, 它将第 1 部分和第 2 部分进行绑定。

sound/soc/codecs/wm9713.c 超过 1300 行, 支持 WM9713、WM9714, 主要定义了一系列的 mixer 控制、DAPM、AC97 底层读写、时钟/PLL 控制以及 snd_soc_ops 的成员函数。我们不可能一一列举, 这里仅抽取其核心一观, 如代码清单 17.35。

代码清单 17.35 WM9713/4 的 ASoC Codec 驱动

```
1  /* 一系列的 mixer 控制 */
2  static const struct soc_enum wm9713_enum[] = {
3  SOC_ENUM_SINGLE(AC97_LINE, 3, 4, wm9713_mic_mixer), /* record mic mixer 0 */
4  ...
5  };
6
7  static const struct snd_kcontrol_new wm9713_snd_ac97_controls[] = {
8  SOC_DOUBLE("Speaker Playback Volume", AC97_MASTER, 8, 0, 31, 1),
9  ...
10 };
11
12 /* add non dapm controls */
13 static int wm9713_add_controls(struct snd_soc_codec *codec)
14 {
15     int err, i;
16
17     for (i = 0; i < ARRAY_SIZE(wm9713_snd_ac97_controls); i++) {
18         err = snd_ctl_add(codec->card,
19             snd_soc_cnew(&wm9713_snd_ac97_controls[i],
20                 codec, NULL));
21         if (err < 0)
22             return err;
23     }
24     return 0;
25 }
26
27 ...
28 static const struct snd_kcontrol_new wm9713_hpl_mixer_controls[] = {
29 ...
30 };
31
32 /* Right Headphone Mixers */
33 static const struct snd_kcontrol_new wm9713_hpr_mixer_controls[] = {
34 ...
35 };
```

```

36
37 /* 一系列的 dapm 控制 */
38 static const struct snd_soc_dapm_widget wm9713_dapm_widgets[] = {
39 SND_SOC_DAPM_MUX("Capture Headphone Mux", SND_SOC_NOPM, 0, 0,
40   &wm9713_hp_rec_mux_controls),
41 ...
42 };
43
44 static const struct snd_soc_dapm_route audio_map[] = {
45 /* left HP mixer */
46 {"Left HP Mixer", "PC Beep Playback Switch", "PCBEEP"},
47 ...
48 };
49
50 static int wm9713_add_widgets(struct snd_soc_codec *codec)
51 {
52   snd_soc_dapm_new_controls(codec, wm9713_dapm_widgets,
53     ARRAY_SIZE(wm9713_dapm_widgets));
54
55   snd_soc_dapm_add_routes(codec, audio_map, ARRAY_SIZE(audio_map));
56
57   snd_soc_dapm_new_widgets(codec);
58   return 0;
59 }
60
61 /* io、时钟、格式等的操作 */
62 static unsigned int ac97_read(struct snd_soc_codec *codec,
63   unsigned int reg)
64 {
65   ...
66 }
67
68 static int ac97_write(struct snd_soc_codec *codec, unsigned int reg,
69   unsigned int val)
70 {
71   ...
72 }
73
74 static int wm9713_set_dai_pll(struct snd_soc_dai *codec_dai,
75   int pll_id, unsigned int freq_in, unsigned int freq_out)
76 {
77   ...
78 }
79
80 static int wm9713_set_dai_fmt(struct snd_soc_dai *codec_dai,
81   unsigned int fmt)
82 {
83   ...
84 }
85
86 /* 关于音频流的操作 snd_soc_ops */
87 static int wm9713_pcm_hw_params(struct snd_pcm_substream *substream,
88   struct snd_pcm_hw_params *params)
89 {
90   ...

```



```
91 }
92
93 static void wm9713_voiceshutdown(struct snd_pcm_substream *substream)
94 {
95     ...
96 }
97
98 static int ac97_hifi_prepare(struct snd_pcm_substream *substream)
99 {
100     ...
101 }
102
103 static int ac97_aux_prepare(struct snd_pcm_substream *substream)
104 {
105     ...
106 }
107
108 /* DAI */
109 struct snd_soc_dai wm9713_dai[] = {
110 {
111     .name = "AC97 HiFi",
112     .type = SND_SOC_DAI_AC97_BUS,
113     .playback = {
114         .stream_name = "HiFi Playback",
115         .channels_min = 1,
116         .channels_max = 2,
117         .rates = WM9713_RATES,
118         .formats = SNDRV_PCM_FMTBIT_S16_LE,},
119     .capture = {
120         .stream_name = "HiFi Capture",
121         .channels_min = 1,
122         .channels_max = 2,
123         .rates = WM9713_RATES,
124         .formats = SNDRV_PCM_FMTBIT_S16_LE,},
125     .ops = {
126         .prepare = ac97_hifi_prepare,},
127     .dai_ops = {
128         .set_clkdiv = wm9713_set_dai_clkdiv,
129         .set_pll = wm9713_set_dai_pll,},
130 },
131 {
132     .name = "AC97 Aux",
133     .playback = {
134         .stream_name = "Aux Playback",
135         .channels_min = 1,
136         .channels_max = 1,
137         .rates = WM9713_RATES,
138         .formats = SNDRV_PCM_FMTBIT_S16_LE,},
139     .ops = {
140         .prepare = ac97_aux_prepare,},
141     .dai_ops = {
142         .set_clkdiv = wm9713_set_dai_clkdiv,
143         .set_pll = wm9713_set_dai_pll,},
144 },
145 }
```

```

146 .name = "WM9713 Voice",
147 ...
148 },
149 };
150 EXPORT_SYMBOL_GPL(wm9713_dai);
151
152 /* snd_soc_codec_device 成员 */
153 static int wm9713_soc_suspend(struct platform_device *pdev,
154 pm_message_t state)
155 {
156 ...
157 }
158
159 static int wm9713_soc_resume(struct platform_device *pdev)
160 {
161 ...
162 }
163
164 static int wm9713_soc_probe(struct platform_device *pdev)
165 {
166 struct snd_soc_device *socdev = platform_get_drvdata(pdev);
167 struct snd_soc_codec *codec;
168 int ret = 0, reg;
169
170 ...
171 codec->name = "WM9713";
172 codec->owner = THIS_MODULE;
173 codec->dai = wm9713_dai;
174 codec->num_dai = ARRAY_SIZE(wm9713_dai);
175 codec->write = ac97_write;
176 codec->read = ac97_read;
177 codec->set_bias_level = wm9713_set_bias_level;
178 INIT_LIST_HEAD(&codec->dapm_widgets);
179 INIT_LIST_HEAD(&codec->dapm_paths);
180
181 ret = snd_soc_new_ac97_codec(codec, &soc_ac97_ops, 0);
182 if (ret < 0)
183     goto codec_err;
184
185 /* register pcms */
186 ret = snd_soc_new_pcms(socdev, SNDRV_DEFAULT_IDX1, SNDRV_DEFAULT_STR1);
187 if (ret < 0)
188     goto pcm_err;
189
190 ...
191
192 wm9713_add_controls(codec);
193 wm9713_add_widgets(codec);
194 ret = snd_soc_register_card(socdev);
195 ...
196 }
197
198 static int wm9713_soc_remove(struct platform_device *pdev)
199 {
200 ...

```



```
201 snd_soc_dapm_free(socdev);
202 snd_soc_free_pcms(socdev);
203 snd_soc_free_ac97_codec(codec);
204 ...
205 return 0;
206 }
207
208 struct snd_soc_codec_device soc_codec_dev_wm9713 = {
209     .probe =      wm9713_soc_probe,
210     .remove =     wm9713_soc_remove,
211     .suspend =    wm9713_soc_suspend,
212     .resume =     wm9713_soc_resume,
213 };
214 EXPORT_SYMBOL_GPL(soc_codec_dev_wm9713);
```

sound/soc/s3c/s3c-ac97.c 实现 CPU 端 AC97 DAI 的驱动, 会导出 `snd_soc_dai` 结构体的实例 `s3c_ac97_dai`。sound/soc/s3c/s3c-pcm.c 实现 CPU 端的 DMA 驱动, 其核心如代码清单 17.36, 它也会导出 `snd_soc_platform` 结构体的实例 `s3c24xx_soc_platform`。

代码清单 17.36 S3C6410 平台驱动 DMA 进行 PCM 流操作

```
1 static const struct snd_pcm硬件 s3c24xx_pcm硬件 = {
2     .info          = SNDRV_PCM_INFO_INTERLEAVED |
3                     SNDRV_PCM_INFO_PAUSE |
4                     SNDRV_PCM_INFO_RESUME |
5                     SNDRV_PCM_INFO_BLOCK_TRANSFER |
6                     SNDRV_PCM_INFO_MMAP |
7                     SNDRV_PCM_INFO_MMAP_VALID,
8     .formats       = SNDRV_PCM_FMTBIT_S16_LE |
9                     SNDRV_PCM_FMTBIT_U16_LE |
10                    SNDRV_PCM_FMTBIT_U8 |
11                    SNDRV_PCM_FMTBIT_S24_LE |
12                    SNDRV_PCM_FMTBIT_S8,
13     .channels_min  = 2,
14     .channels_max  = 2,
15     .buffer_bytes_max = 128*1024,
16     .period_bytes_min = PAGE_SIZE,
17     .period_bytes_max = PAGE_SIZE*2,
18     .periods_min   = 2,
19     .periods_max   = 128,
20     .fifo_size     = 32,
21 };
22
23 static int s3c24xx_pcm_hw_params(struct snd_pcm_substream *substream,
24     struct snd_pcm_hw_params *params)
25 {
26     ...
27 }
28
29 static int s3c24xx_pcm_hw_free(struct snd_pcm_substream *substream)
30 {
31     ...
32 }
33
34 static int s3c24xx_pcm_prepare(struct snd_pcm_substream *substream)
```

```

35 {
36 ...
37 }
38
39 static int s3c24xx_pcm_trigger(struct snd_pcm_substream *substream, int cmd)
40 {
41 ...
42 switch (cmd) {
43 case SNDRV_PCM_TRIGGER_RESUME:
44 ...
45 case SNDRV_PCM_TRIGGER_START:
46 case SNDRV_PCM_TRIGGER_PAUSE_RELEASE:
47     prtd->state |= ST_RUNNING;
48     s3c2410_dma_ctrl(prtd->params->channel, S3C2410_DMAOP_START);
49     break;
50
51 ...
52 }
53 ...
54 }
55
56 static snd_pcm_uframes_t
57 s3c24xx_pcm_pointer(struct snd_pcm_substream *substream)
58 {
59 ...
60 return bytes_to_frames(substream->runtime, res);
61 }
62
63 static int s3c24xx_pcm_open(struct snd_pcm_substream *substream)
64 {
65 ...
66 return 0;
67 }
68
69 static int s3c24xx_pcm_close(struct snd_pcm_substream *substream)
70 {
71 ...
72 }
73
74 static int s3c24xx_pcm_mmap(struct snd_pcm_substream *substream,
75 struct vm_area_struct *vma)
76 {
77 ...
78 return dma_mmap_writecombine(substream->pcm->card->dev, vma,
79                               runtime->dma_area,
80                               runtime->dma_addr,
81                               runtime->dma_bytes);
82 }
83
84 static struct snd_pcm_ops s3c24xx_pcm_ops = {
85 .open      = s3c24xx_pcm_open,
86 .close     = s3c24xx_pcm_close,
87 .ioctl     = snd_pcm_lib_ioctl,
88 .hw_params = s3c24xx_pcm_hw_params,
89 .hw_free   = s3c24xx_pcm_hw_free,

```



```
90 .prepare = s3c24xx_pcm_prepare,
91 .trigger = s3c24xx_pcm_trigger,
92 .pointer = s3c24xx_pcm_pointer,
93 .mmap      = s3c24xx_pcm_mmap,
94 };
95
96 static int s3c24xx_pcm_preallocate_dma_buffer(struct snd_pcm *pcm, int stream)
97 {
98     ...
99     buf->area = dma_alloc_writecombine(pcm->card->dev, size,
100                                       &buf->addr, GFP_KERNEL);
101     if (!buf->area)
102         return -ENOMEM;
103     buf->bytes = size;
104     return 0;
105 }
106
107 static void s3c24xx_pcm_free_dma_buffers(struct snd_pcm *pcm)
108 {
109     ...
110     for(stream=SNDRV_PCM_STREAM_PLAYBACK; stream<=SNDRV_PCM_STREAM_CAPTURE; stream++){
111         substream = pcm->streams[stream].substream;
112         if (!substream)
113             continue;
114
115         buf = &substream->dma_buffer;
116         if (!buf->area)
117             continue;
118
119         dma_free_writecombine(pcm->card->dev, buf->bytes,
120                               buf->area, buf->addr);
121         buf->area = NULL;
122         buf->addr = 0;
123     }
124 }
125
126 static u64 s3c24xx_pcm_dmamask = DMA_32BIT_MASK;
127
128 static int s3c24xx_pcm_new(struct snd_card *card,
129 struct snd_soc_dai *dai, struct snd_pcm *pcm)
130 {
131     ...
132
133     if (dai->playback.channels_min) {
134         ret = s3c24xx_pcm_preallocate_dma_buffer(pcm,
135           SNDRV_PCM_STREAM_PLAYBACK);
136         if (ret)
137             goto out;
138     }
139
140     if (dai->capture.channels_min) {
141         ret = s3c24xx_pcm_preallocate_dma_buffer(pcm,
142           SNDRV_PCM_STREAM_CAPTURE);
143         if (ret)
144             goto out;
```



```

145 }
146 out:
147 return ret;
148 }
149
150 struct snd_soc_platform s3c24xx_soc_platform = {
151     .name      = "s3c24xx-audio",
152     .pcm_ops   = &s3c24xx_pcm_ops,
153     .pcm_new   = s3c24xx_pcm_new,
154     .pcm_free  = s3c24xx_pcm_free_dma_buffers,
155 };
156
157 EXPORT_SYMBOL_GPL(s3c24xx_soc_platform);

```

sound/soc/s3c/smdk6410_wm9713.c 的形式与代码清单 17.33 基本相同，它将前述 Codec 和平台驱动中导出的 symbol 绑定在一起，并在模块加载函数中注册一个名为“soc-audio”的 platform 设备。

17.7 总结

音频设备接口包括 PCM、IIS 和 AC97 等，分别适用于不同的应用场合。针对音频设备，Linux 内核中包含了 3 类音频设备驱动框架，OSS、ALSA 和 ASoC，OSS 包含 dsp 和 mixer 字符设备接口，在用户空间的编程中，完全使用文件操作；ALSA 后者以 card 和组件（PCM、mixer 等）为主线，在用户空间的编程中不使用文件接口而使用 alsalib；ASoC 则是 ALSA 在 SoC 方面的演变，它建立在 ALSA 之上，将 ALSA 驱动中 CPU 相关的代码和 Codec 相关的代码进行了分离。

在音频设备驱动中，几乎必须使用 DMA，而 DMA 的缓冲区会被分割成一个个的段，每次 DMA 操作进行其中的一段。OSS 驱动的阻塞读写具有流控能力，在用户空间不需要进行流量方面的定时工作，但是它需要及时地写（播放）和读（录音），以免出现缓冲区的 underflow 或 overflow。ALSA 和 ASoC 的流控则由 ALSA 的核心层处理，底层驱动仅以 trigger()、pointer() 等方法进行配合。

LINUX

第18章 LCD 设备驱动

在多媒体应用的推动下，彩色 LCD 越来越多地应用到了嵌入式系统中，掌上电脑（PDA）、手机等多采用 TFT 显示器件，支持彩色图形界面，能显示图片并进行视频媒体播放。帧缓冲（Framebuffer）是 Linux 为显示设备提供的一个接口，它允许上层应用程序在图形模式下直接对显示缓冲区分进行读写操作。

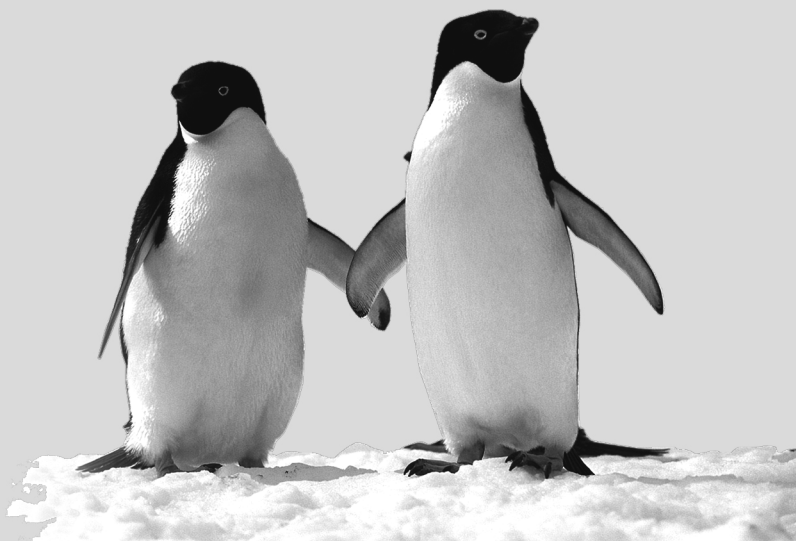
本章主要讲解帧缓冲设备 Linux 驱动的架构及编程方法。

18.1 节讲解了 LCD 的底层硬件操作原理，18.2 节讲解了帧缓冲设备的概念及驱动中的重要数据结构和函数。

18.3 节讲解了帧缓冲设备驱动的整体结构，18.4~18.8 节分别讲解了帧缓冲设备的几个重要函数，18.3 节和 18.4~18.8 节的内容是整体与部分的关系。

18.9 节讲解了 Linux 帧缓冲设备用户空间的访问方法，并对 Qt/Embedded、MiniGUI、MicroWindows、Android 等 GUI 进行了简单的介绍。

18.10 节讲解了 S3C6410 LCD 控制器设备驱动的实例。



18.1 LCD 硬件原理

利用液晶制成的显示器称为 LCD，依据驱动方式可分为静态驱动、简单矩阵驱动以及主动矩阵驱动 3 种。其中，简单矩阵型又可再细分扭转向列型（TN）和超扭转式向列型（STN）两种，而主动矩阵型则以薄膜式晶体管型（TFT）为主流。表 18.1 列出了 TN、STN 和 TFT 显示器的区别。

表 18.1 TN、STN 和 TFT 显示器的区别

类 别	TN	STN	TFT
原理	液晶分子，扭转 90°	扭转 180°~270°	液晶分子，扭转 90°
特性	黑白、单色低对比	黑白、彩色，低对比	彩色(1667 万色)，可媲美 CRT 显示器的全彩，高对比
动画显示	否	否	是
视角	30°以下	40°以下	80°以下
面板尺寸	1~3 英寸	1~12 英寸	37 英寸

TN 型液晶显示技术是 LCD 中最基本的，其他种类的 LCD 都以 TN 型为基础改进而得。TN 型 LCD 显示质量很差，色彩单一，对比度低，反映速度很慢，故主要用于简单的数字符与文字的显示，如电子表及电子计算器等。

STN LCD 的显示原理与 TN 类似，区别在于 TN 型的液晶分子将入射光旋转 90°，而 STN 则可将入射光旋转 180°~270°。STN 改善了 TN 视角狭小的缺点，并提高了对比度，显示品质较 TN 高。

STN 搭配彩色滤光片，将单色显示矩阵的任一像素分成 3 个子像素，分别透过彩色滤光片显示红、绿、蓝三原色，再经由三原色按比例调和，显示出逼近全彩模式的色彩。STN 显示的画面色彩对比度仍较小，反应速度也较慢，可以作为一般的操作显示接口。

随后出现的 DSTN 通过双扫描方式来显示，显示效果相对 STN 而言有了较大幅度的提高。DSTN 的反应速度可达到 100ms，但是在电场反复改变电压的过程中，每一像素的恢复过程较慢。因此，当在屏幕画面快速变化时，会产生“拖尾”现象。

TN 与 STN 型液晶显示器都是使用场电压驱动方式，如果显示尺寸加大，中心部位对电极变化的反应时间就会拉长，显示器的速度跟不上。为了解决这个问题，主动式矩阵驱动被提出，主动式 TFT 型的液晶显示器的结构较为复杂，它包括背光管、导光板、偏光板、滤光板、玻璃基板、配向膜、液晶材料和薄膜式晶体管等。

在 TFT 型 LCD 中，晶体管矩阵依显示信号开启或关闭液晶分子的电压，使液晶分子轴转向而成“亮”或“暗”的对比，避免了显示器对电场效应的依靠。因此，TFT LCD 的显示质量较 TN/STN 更佳，画面显示对比度可达 150:1 以上，反应速度逼近 30ms 甚至更快，适用于 PDA、笔记本电脑、数码相机、MP4 等。

一块 LCD 屏显示图像不但需要 LCD 驱动器，还需要有相应的 LCD 控制器。通常 LCD



驱动器会以 COF/COG 的形式与 LCD 玻璃基板制作在一起, 而 LCD 控制器则由外部电路来实现。许多 MCU 内部直接集成了 LCD 控制器, 通过 LCD 控制器可以方便地控制 STN 和 TFT 屏。

TFT 屏是目前嵌入式系统应用的主流, 图 18.1 所示给出了 TFT 屏的典型时序。时序图中的 VCLK、HSYNC 和 VSYNC 分别为像素时钟信号 (用于锁存图像数据的像素时钟)、行同步信号和帧同步信号, VDEN 为数据有效标志信号, VD 为图像的数据信号。

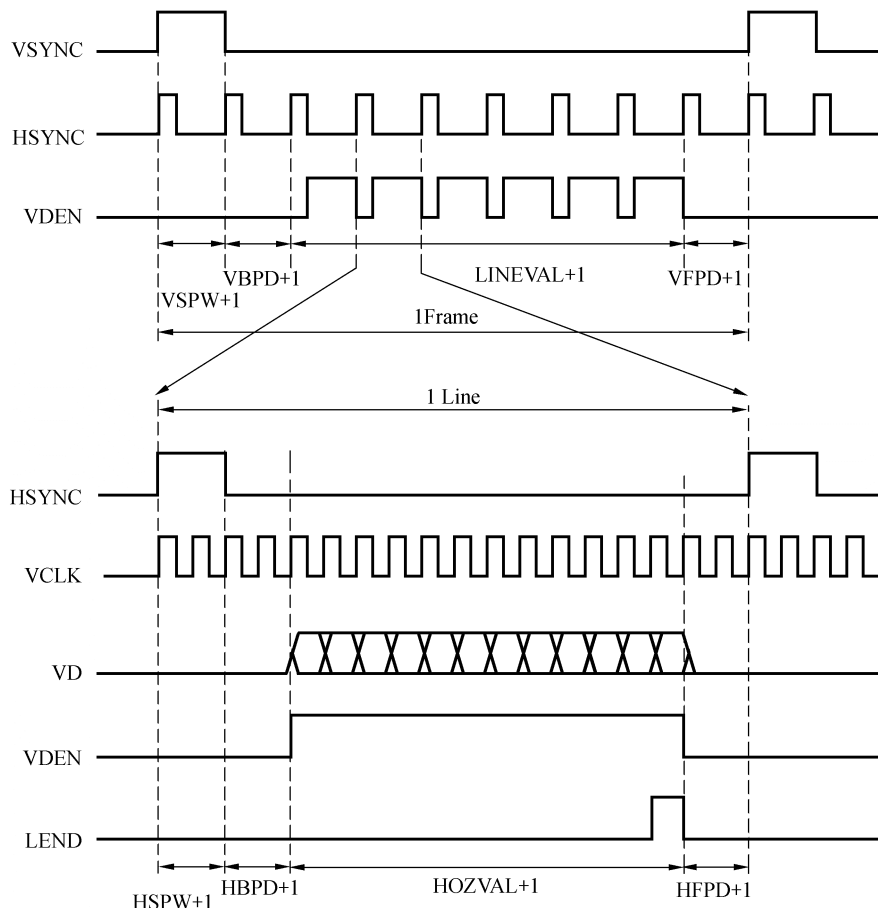


图 18.1 TFT 屏工作时序

作为帧同步信号的 VSYNC, 每发出一个脉冲, 都意味着新的一屏图像数据开始发送。而作为行同步信号的 HSYNC, 每发出一个脉冲都表明新的一行图像资料开始发送。在帧同步以及行同步的头尾都必须留有回扫时间。这样的时序安排起源于 CRT 显示器电子枪偏转所需要的时间, 但后来成为实际上的工业标准, 因此 TFT 屏也包含了回扫时间。

图 18.2 给出了 LCD 控制器中应该设置的 TFT 屏的参数, 其中的上边界和下边界即为帧切换的回扫时间, 左边界和右边界即为行切换的回扫时间, 水平同步和垂直同步分别是行和帧同步本身需要的时间。xres 和 yres 则分别是屏幕的水平和垂直分辨率, 常见的嵌入式设备的 LCD 分辨率主要为 320×240 、 640×480 等。

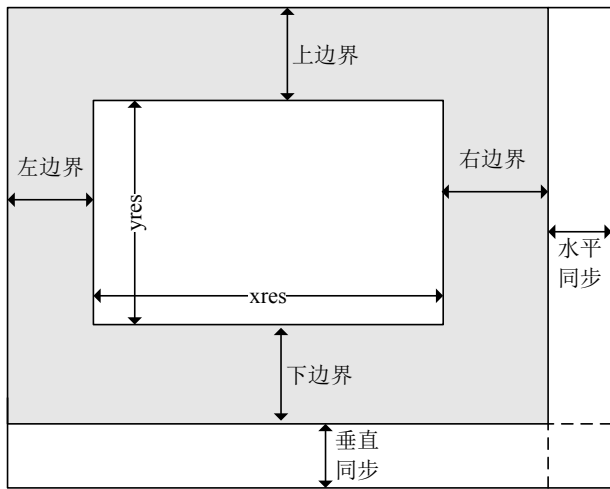


图 18.2 LCD 控制器中的时序参数设置

18.2 帧缓冲

18.2.1 帧缓冲的概念

帧缓冲 (framebuffer) 是 Linux 系统为显示设备提供的一个接口，它将显示缓冲区抽象，屏蔽图像硬件的底层差异，允许上层应用程序在图形模式下直接对显示缓冲区进行读写操作。用户不必关心物理显示缓冲区的具体位置及存放方式，这些都由帧缓冲设备驱动本身来完成。对于帧缓冲设备而言，只要在显示缓冲区中与显示点对应的区域写入颜色值，对应的颜色会自动在屏幕上显示，18.2.2 小节将讲解显示缓冲区与显示点的对应关系。

帧缓冲设备为标准字符设备，主设备号为 29，对应于 `/dev/fbn` 设备文件。帧缓冲驱动的应用非常广泛，在 Linux 的桌面系统中，X Window 服务器就是利用帧缓冲进行窗口的绘制。嵌入式系统中的 Qt/Embedded 等图形用户界面环境也基于帧缓冲而设计。

18.2.2 显示缓冲区与显示点

在帧缓冲设备中，对屏幕显示点的操作通过读写显示缓冲区来完成，在不同的色彩模式下，显示缓冲区和屏幕上的显示点有不同的对应关系，表 18.2~表 18.4 分别给出了 16 级灰度、8 位色和 16 位情况下显示缓冲区与显示点的对应关系。

表 18.2 16 级灰度显示缓冲区与显示点的对应关系

位	31~28	27~24	23~20	19~16	15~12	11~8	7~4	3~0
0x0	点 7	点 6	点 5	点 4	点 3	点 2	点 1	点 0
0x04	点 15	点 14	点 13	点 12	点 11	点 10	点 9	点 8



续表

位	31~28	27~24	23~20	19~16	15~12	11~8	7~4	3~0
...
0x0	点 0	点 1	点 2	点 3	点 4	点 5	点 6	点 7
0x04	点 8	点 9	点 10	点 11	点 12	点 13	点 14	点 15
...

表 18.3 8 位色时显示缓冲区与显示点的对应关系

RGB			BGR		
7~5	4~2	1~0	7~5	4~2	1~0
R	G	B			

表 18.4 16 位色时显示缓冲区与显示点的对应关系

位	15~11		10~5	4~0
RGB565	R		G	B
RGB555		R	G	B

18.2.3 Linux 帧缓冲相关数据结构与函数

1. fb_info 结构体

帧缓冲设备最关键的一个数据结构体是 fb_info 结构体（为了便于记忆，我们把它简称为“FBI”），FBI 中包括了关于帧缓冲设备属性和操作的完整描述，这个结构体的定义如代码清单 18.1 所示。

代码清单 18.1 fb_info 结构体

```
1 struct fb_info {
2     int node;
3     int flags;
4     struct mutex lock;          /* 用于 open/release/ioctl 的锁 */
5     struct fb_var_screeninfo var; /* 可变参数 */
6     struct fb_fix_screeninfo fix; /* 固定参数 */
7     struct fb_monspecs monspecs; /* 显示器标准 */
8     struct work_struct queue; /* 帧缓冲事件队列 */
9     struct fb_pixmap pixmap; /* 图像硬件 mapper */
10    struct fb_pixmap sprite; /* 光标硬件 mapper */
11    struct fb_cmap cmap; /* 目前的颜色表 */
12    struct list_head modelist;
13    struct fb_videomode *mode; /* 目前的 video 模式 */
14
15    #ifdef CONFIG_FB_BACKLIGHT
16        /* 对应的背光设备 */
17        struct backlight_device *bl_dev;
18        /* 背光调整 */
19        struct mutex bl_mutex;
20        u8 bl_curve[FB_BACKLIGHT_LEVELS];
21    #endif
22 }
```

```

22  #ifdef CONFIG_FB_DEFERRED_IO
23      struct delayed_work deferred_work;
24      struct fb_deferred_io *fbdefio;
25  #endif
26  struct fb_ops *fbops; /* fb_ops, 帧缓冲操作 */
27  struct device *device; /* 父设备 */
28  struct device *dev; /* fb 设备 */
29  int class_flag; /* 私有 sysfs 标志 */
30  #ifdef CONFIG_FB_TILEBLITTING
31      struct fb_tile_ops *tileops; /* 图块 Blitting */
32  #endif
33  char _iomem *screen_base; /* 虚拟基地址 */
34  unsigned long screen_size; /* ioremapped 的虚拟内存大小 */
35  void *pseudo_palette; /* 伪 16 色颜色表 */
36  #define FBINFO_STATE_RUNNING 0
37  #define FBINFO_STATE_SUSPENDED 1
38  u32 state; /* 硬件状态, 如挂起 */
39  void *fbcon_par;
40  void *par;
41 };

```

FBI 中记录了帧缓冲设备的全部信息, 包括设备的设置参数、状态以及操作函数指针。每一个帧缓冲设备都必须对应一个 FBI。

2. fb_ops 结构体

FBI 的成员变量 fbops 为指向底层操作的函数的指针, 这些函数是需要驱动程序开发人员编写的, 其定义如代码清单 18.2 所示。

代码清单 18.2 fb_ops 结构体

```

1  struct fb_ops {
2      struct module *owner;
3      /* 打开/释放 */
4      int(*fb_open)(struct fb_info *info, int user);
5      int(*fb_release)(struct fb_info *info, int user);
6
7      /* 对于非线性布局的/常规内存映射无法工作的帧缓冲设备需要 */
8      ssize_t(*fb_read)(struct file *file, char __user *buf, size_t count,
9          loff_t *ppos);
10     ssize_t(*fb_write)(struct file *file, const char __user *buf, size_t count,
11         loff_t *ppos);
12
13     /* 检测可变参数, 并调整到支持的值 */
14     int(*fb_check_var)(struct fb_var_screeninfo *var, struct fb_info *info);
15
16     /* 根据 info->var 设置 video 模式 */
17     int(*fb_set_par)(struct fb_info *info);
18
19     /* 设置 color 寄存器 */
20     int(*fb_setcolreg)(unsigned regno, unsigned red, unsigned green, unsigned
21         blue, unsigned transp, struct fb_info *info);
22
23     /* 批量设置 color 寄存器, 设置颜色表 */
24     int(*fb_setcmap)(struct fb_cmap *cmap, struct fb_info *info);
25

```



```
26  /*显示空白 */
27  int(*fb_blank)(int blank, struct fb_info *info);
28
29  /* pan 显示 */
30  int(*fb_pan_display)(struct fb_var_screeninfo *var, struct fb_info *info);
31
32  /* 矩形填充 */
33  void(*fb_fillrect)(struct fb_info *info, const struct fb_fillrect *rect);
34  /* 数据复制 */
35  void(*fb_copyarea)(struct fb_info *info, const struct fb_copyarea *region);
36  /* 图形填充 */
37  void(*fb_imageblit)(struct fb_info *info, const struct fb_image *image);
38
39  /* 绘制光标 */
40  int(*fb_cursor)(struct fb_info *info, struct fb_cursor *cursor);
41
42  /* 旋转显示 */
43  void(*fb_rotate)(struct fb_info *info, int angle);
44
45  /* 等待 blit 空闲 (可选) */
46  int(*fb_sync)(struct fb_info *info);
47
48  /* fb 特定的 ioctl (可选) */
49  int(*fb_ioctl)(struct fb_info *info, unsigned int cmd, unsigned long arg);
50
51  /* 处理 32 位的 compat ioctl (可选) */
52  int(*fb_compat_ioctl)(struct fb_info *info, unsigned cmd, unsigned long arg);
53
54  /* fb 特定的 mmap */
55  int(*fb_mmap)(struct fb_info *info, struct vm_area_struct *vma);
56
57  /* 保存目前的硬件状态 */
58  void(*fb_save_state)(struct fb_info *info);
59
60  /* 恢复被保存的硬件状态 */
61  void(*fb_restore_state)(struct fb_info *info);
62
63  void (*fb_get_caps)(struct fb_info *info, struct fb_blit_caps *caps,
64                    struct fb_var_screeninfo *var);
65 };
```

fb_ops 的 fb_check_var()成员函数用于检查可以修改的屏幕参数并调整到合适的值, 而 fb_set_par()则使得用户设置的屏幕参数在硬件上有效。

3. fb_var_screeninfo 和 fb_fix_screeninfo 结构体

FBI 的 fb_var_screeninfo 和 fb_fix_screeninfo 成员也是结构体, fb_var_screeninfo 记录用户可修改的显示控制器参数, 包括屏幕分辨率和每个像素点的比特数。fb_var_screeninfo 中的 xres 定义屏幕一行有多少个点, yres 定义屏幕一列有多少个点, bits_per_pixel 定义每个点用多少个字节表示。而 fb_fix_screeninfo 中记录用户不能修改的显示控制器的参数, 如屏幕缓冲区的物理地址、长度。当对帧缓冲设备进行映射操作的时候, 就是从 fb_fix_screeninfo 中取得缓冲区物理地址的。上述数据成员都需要在驱动程序中初始化和设置。

fb_var_screeninfo 和 fb_fix_screeninfo 结构体的定义分别如代码清单 18.3 和代码清单 18.4 所示。

代码清单 18.3 fb_var_screeninfo 结构体

```

1 struct fb_var_screeninfo {
2     /* 可见解析度 */
3     u32 xres;
4     u32 yres;
5     /* 虚拟解析度 */
6     u32 xres_virtual;
7     u32 yres_virtual;
8     /* 虚拟到可见之间的偏移 */
9     u32 xoffset;
10    u32 yoffset;
11
12    u32 bits_per_pixel; /* 每像素位数, BPP */
13    u32 grayscale; /* 非 0 时指灰度 */
14
15    /* fb 缓存的 R\G\B 位域 */
16    struct fb_bitfield red;
17    struct fb_bitfield green;
18    struct fb_bitfield blue;
19    struct fb_bitfield transp; /* 透明度 */
20
21    u32 nonstd; /* != 0 非标准像素格式 */
22
23    u32 activate;
24
25    u32 height; /* 高度 */
26    u32 width; /* 宽度 */
27
28    u32 accel_flags; /* 看 fb_info.flags */
29
30    /* 定时: 除了 pixclock 本身外, 其他的都以像素时钟为单位 */
31    u32 pixclock; /* 像素时钟 (皮秒) */
32    u32 left_margin; /* 行切换: 从同步到绘图之间的延迟 */
33    u32 right_margin; /* 行切换: 从绘图到同步之间的延迟 */
34    u32 upper_margin; /* 帧切换: 从同步到绘图之间的延迟 */
35    u32 lower_margin; /* 帧切换: 从绘图到同步之间的延迟 */
36    u32 hsync_len; /* 水平同步的长度 */
37    u32 vsync_len; /* 垂直同步的长度 */
38    u32 sync;
39    u32 vmode;
40    u32 rotate; /* 顺时针旋转的角度 */
41    u32 reserved[5]; /* 保留 */
42 };

```

代码清单 18.4 fb_fix_screeninfo 结构体

```

1 struct fb_fix_screeninfo {
2     char id[16]; /* 字符串形式的标识符 */
3     unsigned long smem_start; /* fb 缓冲内存的开始位置 (物理地址) */
4     u32 smem_len; /* fb 缓冲的长度 */
5     u32 type; /* FB_TYPE_* */
6     u32 type_aux; /* Interleave */
7     u32 visual; /* FB_VISUAL_* */
8     u16 xpanstep; /* 如果没有硬件 panning, 赋 0 */

```



```
9    u16 ypanstep;
10   u16 ywrapstep;
11   u32 line_length; /* 1 行的字节数 */
12   unsigned long mmio_start; /* 内存映射 I/O 的开始位置 */
13   u32 mmio_len; /* 内存映射 I/O 的长度 */
14   u32 accel;
15   u16 reserved[3]; /* 保留 */
16 };
```

代码清单 18.4 中第 7 行的 `visual` 记录屏幕使用的色彩模式, 在 Linux 系统中, 支持的色彩模式包括如下几种。

- Monochrome (FB_VISUAL_MONO01、FB_VISUAL_MONO10), 每个像素是黑或白。
- Pseudo color (FB_VISUAL_PSEUDOCOLOR、FB_VISUAL_STATIC_PSEUDOCOLOR), 即伪彩色, 采用索引颜色显示。
- True color (FB_VISUAL_TRUECOLOR), 真彩色, 分成红、绿、蓝三基色。
- Direct color (FB_VISUAL_DIRECTCOLOR), 每个像素颜色也是由红、绿、蓝组成, 不过每个颜色值是个索引, 需要查表。
- Grayscale displays, 灰度显示, 红、绿、蓝的值都一样。

4. fb_bitfield 结构体

代码清单 18.3 第 16、17、18 行分别记录 R、G、B 的位域, `fb_bitfield` 结构体描述每一像素显示缓冲区的组织方式, 包含位域偏移、位域长度和 MSB (最高有效位) 指示, 如代码清单 18.5 所示。

代码清单 18.5 fb_bitfield 结构体

```
1 struct fb_bitfield {
2     __u32 offset; /* 位域偏移 */
3     __u32 length; /* 位域长度 */
4     __u32 msb_right; /* !=0: MSB 在右边 */
5 };
```

5. fb_cmap 结构体

`fb_cmap` 结构体记录设备无关的颜色表信息, 用户空间可以通过 `ioctl()` 的 `FBIOGETCMAP` 和 `FBIOPUTCMAP` 命令读取或设定颜色表。

代码清单 18.6 fb_cmap 结构体

```
1 struct fb_cmap {
2     u32 start; /* 第 1 个元素入口 */
3     u32 len; /* 元素数量 */
4     /* R、G、B、透明度 */
5     u16 *red;
6     u16 *green;
7     u16 *blue;
8     u16 *transp;
9 };
```

代码清单 18.7 所示为用户空间获取颜色表的例程, 若 BPP 为 8 位, 则颜色表长度为 256; 若 BPP 为 4 位, 则颜色表长度为 16; 否则, 颜色表长度为 0, 这是因为, 对于 BPP 大于等于 16 的情况, 使用颜色表是不划算的。

代码清单 18.7 用户空间获取颜色表例程

```

1  /* 读入颜色表 */
2  if ((vinfo.bits_per_pixel == 8) || (vinfo.bits_per_pixel == 4)) {
3      screencols = (vinfo.bits_per_pixel == 8) ? 256 : 16; /* 颜色表大小 */
4      int loopc;
5      startcmap = new fb_cmap;
6      startcmap->start = 0;
7      startcmap->len = screencols;
8      /* 分配颜色表的内存 */
9      startcmap->red = (unsigned short int*)malloc(sizeof(unsigned short int)
10         *screencols);
11      startcmap->green = (unsigned short int*)malloc(sizeof(unsigned short int)
12         *screencols);
13      startcmap->blue = (unsigned short int*)malloc(sizeof(unsigned short int)
14         *screencols);
15      startcmap->transp = (unsigned short int*)malloc(sizeof(unsigned short int)
16         *screencols);
17      /* 获取颜色表 */
18      ioctl(fd, FBIOGETCMAP, startcmap);
19      for (loopc = 0; loopc < screencols; loopc++) {
20          screenclut[loopc] = qRgb(startcmap->red[loopc] >> 8, startcmap
21             ->green[loopc] >> 8, startcmap->blue[loopc] >> 8);
22      }
23 }

```

对于一个 256 色 (BPP=8) 的 800×600 分辨率的图像而言, 若红、绿、蓝分别用一个字节描述, 则需要 $800 \times 600 \times 3 = 1\,440\,000$ Byte 的空间, 而若使用颜色表, 则只需要 $800 \times 600 \times 1 + 256 \times 3 = 480\,768$ Byte 的空间。

6. 文件操作结构体

作为一种字符设备, 帧缓冲设备的文件操作结构体定义于 `/linux/drivers/vedio/fbmem.c` 文件中, 如代码清单 18.8 所示。

代码清单 18.8 帧缓冲设备文件操作结构体

```

1  static struct file_operations fb_fops = {
2      .owner = THIS_MODULE,
3      .read = fb_read,
4      .write = fb_write,
5      .ioctl = fb_ioctl,
6      #ifdef CONFIG_COMPAT
7          .compat_ioctl = fb_compat_ioctl,
8      #endif
9      .mmap = fb_mmap,
10     .open = fb_open,
11     .release = fb_release,
12     #ifdef HAVE_ARCH_FB_UNMAPPED_AREA
13         .get_unmapped_area = get_fb_unmapped_area,
14     #endif
15     #ifdef CONFIG_FB_DEFERRED_IO
16         .fsync = fb_deferred_io_fsync,
17     #endif
18 };

```



帧缓冲设备驱动的文件操作接口函数已经在 fbmem.c 中被统一实现，一般不需要由驱动工程师再编写。

7. 注册与注销帧缓冲设备

Linux 内核提供了 register_framebuffer() 和 unregister_framebuffer() 函数分别注册和注销帧缓冲设备，这两个函数都接受 FBI 指针为参数，原型为：

```
int register_framebuffer(struct fb_info *fb_info);  
int unregister_framebuffer(struct fb_info *fb_info);
```

对于 register_framebuffer() 函数而言，如果注册的帧缓冲设备数超过了 FB_MAX（目前定义为 32），则函数返回 -ENXIO，注册成功则返回 0。

18.3 Linux 帧缓冲设备驱动结构

图 18.3 所示为 Linux 帧缓冲设备驱动的主要结构，帧缓冲设备提供给用户空间的 file_operations 结构体由 fbmem.c 中的 file_operations 提供，而特定帧缓冲设备 fb_info 结构体的注册、注销以及其中成员的维护，尤其是 fb_ops 中成员函数的实现则由对应的 xxxfb.c 文件实现，fb_ops 中的成员函数最终会操作 LCD 控制器硬件寄存器。

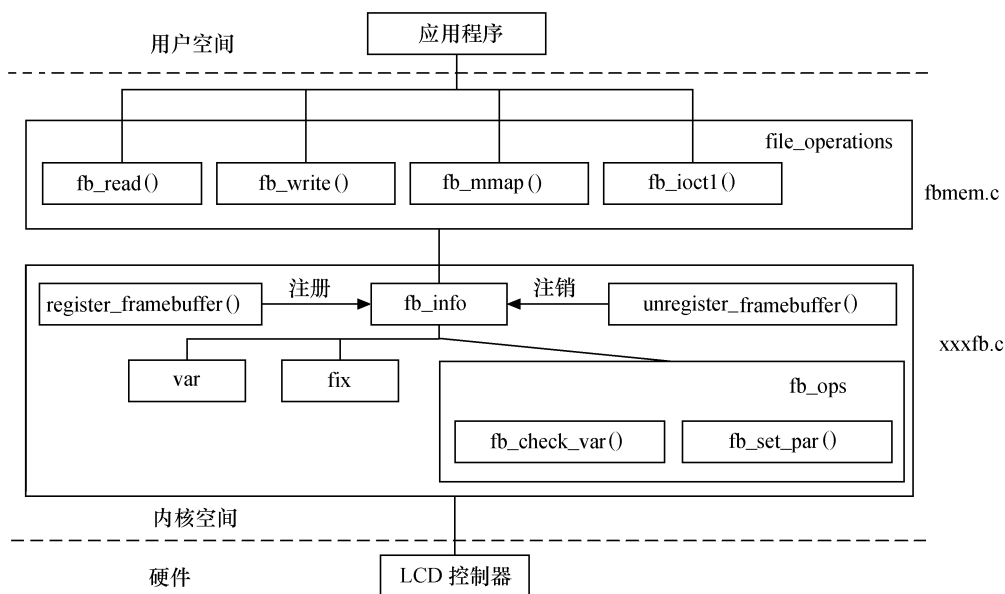


图 18.3 帧缓冲设备驱动的程序结构

18.4 帧缓冲设备驱动模块加载与卸载函数

在帧缓冲设备驱动模块加载函数中，应该完成如下 4 个工作。

(1) 申请 FBI 结构体的内存空间，初始化 FBI 结构体中固定和可变的屏幕参数，即填充 FBI 中 `fb_var_screeninfo` `var` 和 `struct fb_fix_screeninfo` `fix` 成员。

(2) 根据具体 LCD 屏幕的特点，完成 LCD 控制器硬件的初始化。

(3) 申请帧缓冲设备的显示缓冲区空间。

(4) 注册帧缓冲设备。

在帧缓冲设备驱动的模块卸载函数中，应该完成相反的工作，包括释放 FBI 结构体内存、关闭 LCD、释放显示缓冲区以及注销帧缓冲设备。

由于 LCD 控制器经常被集成在 SoC 上作为一个独立的硬件模块而存在（成为 `platform_device`），因此，LCD 驱动中也经常包含平台驱动，这样，在帧缓冲设备驱动的模块加载函数中完成的工作只是注册平台驱动，而初始化 FBI 结构体中的固定和可变参数、LCD 控制器硬件的初始化、申请帧缓冲设备的显示缓冲区空间和注册帧缓冲设备的工作则移交到平台驱动的探测函数中完成。

同样地，在使用平台驱动的情况下，释放 FBI 结构体内存、关闭 LCD、释放显示缓冲区以及注销帧缓冲设备的工作也移交到平台驱动的移除函数中完成。

代码清单 18.9 所示为帧缓冲设备驱动的模块加载和卸载以及平台驱动的探测和移除函数中的模板。

代码清单 18.9 帧缓冲设备驱动的模块加载/卸载及平台驱动的探测/移除函数的模板

```

1  /* 平台驱动结构体 */
2  static struct platform_driver xxxfb_driver = {
3      .probe = xxxfb_probe,
4      .remove = xxxfb_remove,
5      .suspend = xxxfb_suspend,
6      .resume = xxxfb_resume,
7      .driver = {
8          .name = "xxx-lcd", /* 驱动名 */
9          .owner = THIS_MODULE,
10     }
11 };
12
13 /* 平台驱动探测函数 */
14 static int __init xxxfb_probe(...)
15 {
16     struct fb_info *info;
17
18     /*分配 fb_info 结构体*/
19     info = framebuffer_alloc(...);
20
21     info->screen_base = framebuffer_virtual_memory;
22     info->var = xxxfb_var; /* 可变参数 */
23     info->fix = xxxfb_fix; /* 固定参数 */
24
25     /*分配显示缓冲区*/
26     alloc_dis_buffer(...);
27
28     /*初始化 LCD 控制器*/
29     lcd_init(...);
30
31     /*检查可变参数*/
32     xxxfb_check_var(&info->var, info);

```



```
33
34  /*注册 fb_info*/
35  if (register_framebuffer(info) < 0)
36      return -EINVAL;
37
38  return 0;
39 }
40
41 /* 平台驱动移除函数 */
42 static void __exit xxxfb_remove(...)
43 {
44     struct fb_info *info = dev_get_drvdata(dev);
45
46     if (info) {
47         unregister_framebuffer(info); /* 注销 fb_info */
48         dealloc_dis_buffer(...); /* 释放显示缓冲区 */
49         framebuffer_release(info); /* 注销 fb_info */
50     }
51
52     return 0;
53 }
54
55 /* 帧缓冲设备驱动模块加载与卸载函数 */
56 int __init xxxfb_init(void)
57 {
58     return platform_driver_register(&xxxfb_driver); /* 注册平台设备 */
59 }
60
61 static void __exit xxxfb_cleanup(void)
62 {
63     platform_driver_unregister(&xxxfb_driver); /* 注销平台设备 */
64 }
65
66 module_init(xxxfb_init);
67 module_exit(xxxfb_cleanup);
```

上述代码中第 35 行、47 行成对出现的 `register_framebuffer()` 和 `unregister_framebuffer()` 分别用于注册和注销帧缓冲设备。

18.5 帧缓冲设备显示缓冲区的申请与释放

在嵌入式系统中，一种常见的方式是直接在 RAM 空间中分配一段显示缓冲区，典型结构如图 18.4 所示。

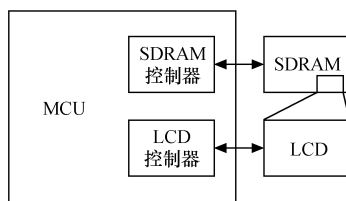


图 18.4 在 RAM 中分配显示缓冲区



在分配显示缓冲区时一定要考虑 cache 的一致性问题,因为系统往往会通过 DMA 方式搬移显示数据。合适的方式是使用 `dma_alloc_writecombine()` 函数分配一段 `writecombining` 区域,对应的 `writecombining` 区域由 `dma_free_writecombine()` 函数释放,如代码清单 18.10 所示。

`writecombining` 意味着“写合并”,它允许写入的数据被合并,并临时保存在写合并缓冲区(WCB)中,直到进行一次 burst 传输而不再需要多次 single 传输。通过 `dma_alloc_writecombine()` 分配的显示缓冲区不会出现 cache 一致性问题,这一点类似于 `dma_alloc_coherent()`。

代码清单 18.10 帧缓冲设备显示缓冲区的分配与释放

```
1 static int __init xxxfb_map_video_memory(struct xxxfb_info *fbi)
2 {
3     fbi->map_size = PAGE_ALIGN(fbi->fb->fix.smem_len + PAGE_SIZE);
4     fbi->map_cpu = dma_alloc_writecombine(fbi->dev, fbi->map_size,
5         &fbi->map_dma, GFP_KERNEL); /* 分配内存 */
6
7     fbi->map_size = fbi->fb->fix.smem_len; /* 显示缓冲区大小 */
8
9     if (fbi->map_cpu) {
10         memset(fbi->map_cpu, 0xf0, fbi->map_size);
11
12         fbi->screen_dma = fbi->map_dma;
13         fbi->fb->screen_base = fbi->map_cpu;
14         fbi->fb->fix.smem_start = fbi->screen_dma;
15     }
16
17     return fbi->map_cpu ? 0 : - ENOMEM;
18 }
19
20 static inline void xxxfb_unmap_video_memory(struct s3c2410fb_info *fbi)
21 {
22     /* 释放显示缓冲区 */
23     dma_free_writecombine(fbi->dev, fbi->map_size, fbi->map_cpu, fbi->map_dma);
24 }
```

18.6 帧缓冲设备的参数设置

18.6.1 定时参数

FBI 结构体可变参数 `var` 中的 `left_margin`、`right_margin`、`upper_margin`、`lower_margin`、`hsync_len` 和 `vsync_len` 直接查 LCD 的数据手册就可以得到,图 18.5 所示为某 LCD 数据手册中直接抓图获得的定时信息。

由图 18.5 可知对该 LCD 而言, `var` 中各参数的较合适值分别为: `left_margin` = 104, `right_margin` = 8, `upper_margin` = 16, `lower_margin` = 2, `hsync_len` = 8, `vsync_len` = 2。



显示模式	参数	符号	条件	时间			单位
				最小值	典型值	最大值	
常规	Vertical cvcle	VP		648	660	670	行
	Vertical data start	VDS	VS+VBP	4	4	4	行
	Vertical Sync Pulse width	VS		2	2	2	行
	Vertical front porch	VFP		4	16	26	行
	Vertical Back porch	VBP		2	2	2	行
	Vertical blanking period	VBL	VS+VBP+VFP	8	20	30	行
	Vertical active area	VDISP		640	640	640	行
	Horizontal cycle	HP		559	600	620	点
	Horizontal front porch	HFP		63	104	124	点
	Horizontal Sync Pulse width	HS		8	8	8	点
	Horizontal Back porch	HBP		8	8	8	点
	Horizontal Data start	HDS	HS+HBP	16	16	16	点
	Horizontal active area	HDISP		480	480	480	点
	Clock frequency	fclk		22	26	28	MHz
		tcclk		45	38	35	ns

图 18.5 LCD 数据手册中定时参数示例

18.6.2 像素时钟

FBI 可变参数 var 中的 pixclock 意味着像素时钟，例如，如果为 28.37516 MHz，那么画 1 个像素需要 35242 ps（皮秒）：

```
1/(28.37516E6 Hz) = 35.242E-9 s
```

如果屏幕的分辨率是 640×480，显示一行需要的时间是：

```
640*35.242E-9 s = 22.555E-6 s
```

每条扫描线是 640，但是水平回扫和水平同步也需要时间，假设水平回扫和同步需要 272 个像素时钟，因此，画一条扫描线完整的时间是：

```
(640+272)*35.242E-9 s = 32.141E-6 s
```

可以计算出水平扫描率大约是 31kHz：

```
1/(32.141E-6 s) = 31.113E3 Hz
```

完整的屏幕有 480 线，但是垂直回扫和垂直同步也需要时间，假设垂直回扫和垂直同步需要 49 个像素时钟，因此，画一个完整的屏幕的时间是：

```
(480+49)*32.141E-6 s = 17.002E-3 s
```

可以计算出垂直扫描率大约是 59kHz：

```
1/(17.002E-3 s) = 58.815 Hz
```

这意味着屏幕数据每秒钟大约刷新 59 次。

18.6.3 颜色位域

FBI 可变参数 var 中的 red、green 和 blue 位域的设置直接由显示缓冲区与显示点的对应关系决定，例如，对于 RGB565 模式，查表 18.4，red 占据 5 位，偏移为 11 位；green 占据 6 位，偏移为 5 位；blue 占据 5 位，偏移为 0 位，即：

```
fbinfo->var.red.offset = 11;
fbinfo->var.green.offset = 5;
fbinfo->var.blue.offset = 0;
fbinfo->var.transp.offset = 0;
fbinfo->var.red.length = 5;
fbinfo->var.green.length = 6;
```



```
fbinfo->var.blue.length = 5;
```

18.6.4 固定参数

FBI 固定参数 `fix` 中的 `smem_start` 指示帧缓冲设备显示缓冲区的首地址，`smem_len` 为帧缓冲设备显示缓冲区的大小，计算公式为：

```
smem_len = max_xres * max_yres * max_bpp
```

即：帧缓冲设备显示缓冲区的大小 = 最大的 x 解析度 * 最大的 y 解析度 * 最大的 BPP。

18.7 帧缓冲设备驱动的 fb_ops 成员函数

FBI 中的 `fb_ops` 是使得帧缓冲设备工作所需函数的集合，它们最终与 LCD 控制器硬件打交道。

`fb_check_var()` 用于调整可变参数，并修正为硬件所支持的值；`fb_set_par()` 则根据屏幕参数设置具体读写 LCD 控制器的寄存器以使得 LCD 控制器进入相应的工作状态。

对于 `fb_ops` 中的 `fb_fillrect()`、`fb_copyarea()` 和 `fb_imageblit()` 成员函数，通常直接使用对应的通用的 `cfb_fillrect()`、`cfb_copyarea()` 和 `cfb_imageblit()` 函数即可。`cfb_fillrect()` 函数定义在 `drivers/video/cfbfillrect.c` 文件中，`cfb_copyarea()` 定义在 `drivers/video/cfbcopyarea.c` 文件中，`cfb_imageblit()` 定义在 `drivers/video/cfbimgblt.c` 文件中。

`fb_ops` 中的 `fb_setcolreg()` 成员函数实现伪颜色表（针对 `FB_VISUAL_TRUECOLOR`、`FB_VISUAL_DIRECTCOLOR` 模式）和颜色表的填充，其模板如代码清单 18.11 所示。

代码清单 18.11 `fb_setcolreg()` 函数模板

```
1 static int xxxfb_setcolreg(unsigned regno, unsigned red, unsigned green,
2   unsigned blue, unsigned transp, struct fb_info *info)
3 {
4     struct xxxfb_info *fbi = info->par;
5     unsigned int val;
6
7     switch (fbi->fb->fix.visual) {
8     case FB_VISUAL_TRUECOLOR:
9         /* 真彩色，设置伪颜色表 */
10        if (regno < 16) {
11            u32 *pal = fbi->fb->pseudo_palette;
12
13            val = chan_to_field(red, &fbi->fb->var.red);
14            val |= chan_to_field(green, &fbi->fb->var.green);
15            val |= chan_to_field(blue, &fbi->fb->var.blue);
16
17            pal[regno] = val;
18        }
19        break;
20
21     case FB_VISUAL_PSEUDOCOLOR:
22        if (regno < 256) {
23            /* RGB565 模式 */
24            val = ((red >> 0) & 0xf800);
```



```
25         val |= ((green >> 5) &0x07e0);
26         val |= ((blue >> 11) &0x001f);
27
28         writel(val, XXX_TFTPAL(regno));
29         schedule_palette_update(fbi, regno, val);
30     }
31     break;
32     ...
33 }
34
35 return 0;
36 }
```

上述代码第 10 行对 `regno < 16` 的判断意味着伪颜色表只有 16 个成员，实际上，它们对应 16 种控制台颜色，logo 显示也会使用伪颜色表。

18.8 LCD 设备驱动的读写、mmap 和 ioctl 函数

虽然帧缓冲设备的 `file_operations` 中的成员函数，即文件操作函数已经由内核在 `fbmem.c` 文件中实现，一般不再需要驱动工程师修改，但分析这些函数对于巩固字符设备驱动的知识以及加深对帧缓冲设备驱动的理解是大有裨益的。

代码清单 18.12 所示为 LCD 设备驱动的文件操作读写函数的源代码，从代码结构及习惯而言，与本书第 2 篇所讲解的字符设备驱动完全一致。

代码清单 18.12 帧缓冲设备驱动的读写函数

```
1 static ssize_t
2 fb_read(struct file *file, char __user *buf, size_t count, loff_t *ppos)
3 {
4     unsigned long p = *ppos;
5     struct inode *inode = file->f_path.dentry->d_inode;
6     int fbidx = iminor(inode);
7     struct fb_info *info = registered_fb[fbidx];
8     u32 *buffer, *dst;
9     u32 __iomem *src;
10    int c, i, cnt = 0, err = 0;
11    unsigned long total_size;
12
13    ...
14
15    buffer = kmalloc((count > PAGE_SIZE) ? PAGE_SIZE : count,
16                    GFP_KERNEL);
17    if (!buffer)
18        return -ENOMEM;
19
20    src = (u32 __iomem *) (info->screen_base + p);
21
22    if (info->fbops->fb_sync)
23        info->fbops->fb_sync(info);
24
25    while (count) {
```

```

26     c = (count > PAGE_SIZE) ? PAGE_SIZE : count;
27     dst = buffer;
28     for (i = c >> 2; i--; )
29         *dst++ = fb_readl(src++);
30     if (c & 3) {
31         u8 *dst8 = (u8 *) dst;
32         u8 __iomem *src8 = (u8 __iomem *) src;
33
34         for (i = c & 3; i--;)
35             *dst8++ = fb_readb(src8++);
36
37         src = (u32 __iomem *) src8;
38     }
39
40     if (copy_to_user(buf, buffer, c)) {
41         err = -EFAULT;
42         break;
43     }
44     *ppos += c;
45     buf += c;
46     cnt += c;
47     count -= c;
48 }
49
50 kfree(buffer);
51
52 return (err) ? err : cnt;
53 }
54
55 static ssize_t
56 fb_write(struct file *file, const char __user *buf, size_t count, loff_t *ppos)
57 {
58     unsigned long p = *ppos;
59     struct inode *inode = file->f_path.dentry->d_inode;
60     int fbidx = iminor(inode);
61     struct fb_info *info = registered_fb[fbidx];
62     u32 *buffer, *src;
63     u32 __iomem *dst;
64     int c, i, cnt = 0, err = 0;
65     unsigned long total_size;
66
67     ...
68
69     buffer = kmalloc((count > PAGE_SIZE) ? PAGE_SIZE : count,
70                     GFP_KERNEL);
71     if (!buffer)
72         return -ENOMEM;
73
74     dst = (u32 __iomem *) (info->screen_base + p);
75
76     if (info->fbops->fb_sync)
77         info->fbops->fb_sync(info);
78
79     while (count) {
80         c = (count > PAGE_SIZE) ? PAGE_SIZE : count;

```



```
81     src = buffer;
82
83     if (copy_from_user(src, buf, c)) {
84         err = -EFAULT;
85         break;
86     }
87
88     for (i = c >> 2; i--; )
89         fb_writel(*src++, dst++);
90
91     if (c & 3) {
92         u8 *src8 = (u8 *) src;
93         u8 __iomem *dst8 = (u8 __iomem *) dst;
94
95         for (i = c & 3; i--; )
96             fb_writeb(*src8++, dst8++);
97
98         dst = (u32 __iomem *) dst8;
99     }
100
101     *ppos += c;
102     buf += c;
103     cnt += c;
104     count -= c;
105 }
106
107 kfree(buffer);
108
109 return (cnt) ? cnt : err;
110 }
```

`file_operations` 中的 `mmap()` 函数非常关键, 它将显示缓冲区映射到用户空间, 从而使得用户空间可以直接操作显示缓冲区而省去一次用户空间到内核空间的内存复制过程, 提高效率, 其源代码如代码清单 18.13 所示。

代码清单 18.13 帧缓冲设备驱动的 `mmap` 函数

```
1 static int
2 fb_mmap(struct file *file, struct vm_area_struct * vma)
3 __acquires(&info->lock)
4 __releases(&info->lock)
5 {
6     int fbidx = iminor(file->f_path.dentry->d_inode);
7     struct fb_info *info = registered_fb[fbidx];
8     struct fb_ops *fb = info->fbops;
9     unsigned long off;
10    unsigned long start;
11    u32 len;
12
13    if (vma->vm_pgoff > (~0UL >> PAGE_SHIFT))
14        return -EINVAL;
15    off = vma->vm_pgoff << PAGE_SHIFT;
16    if (!fb)
17        return -ENODEV;
18    if (fb->fb_mmap) {
```

```

19     int res;
20     mutex_lock(&info->lock);
21     res = fb->fb_mmap(info, vma);
22     mutex_unlock(&info->lock);
23     return res;
24 }
25
26 mutex_lock(&info->lock);
27
28 /* 帧缓冲区内存 */
29 start = info->fix.smem_start;
30 len = PAGE_ALIGN((start & ~PAGE_MASK) + info->fix.smem_len);
31 if (off >= len) {
32     /* 内存映射的 IO */
33     off -= len;
34     if (info->var.accel_flags) {
35         mutex_unlock(&info->lock);
36         return -EINVAL;
37     }
38     start = info->fix.mmio_start;
39     len = PAGE_ALIGN((start & ~PAGE_MASK) + info->fix.mmio_len);
40 }
41 mutex_unlock(&info->lock);
42 start &= PAGE_MASK;
43 if ((vma->vm_end - vma->vm_start + off) > len)
44     return -EINVAL;
45 off += start;
46 vma->vm_pgoff = off >> PAGE_SHIFT;
47 /* 这是一个 I/O 映射 - 告诉 maydump 跳过此 VMA */
48 vma->vm_flags |= VM_IO | VM_RESERVED;
49 fb_pgprotect(file, vma, off);
50 if (io_remap_pfn_range(vma, vma->vm_start, off >> PAGE_SHIFT,
51     vma->vm_end - vma->vm_start, vma->vm_page_prot))
52     return -EAGAIN;
53 return 0;
54 }

```

`fb_ioctl()` 函数最终实现对用户 I/O 控制命令的执行，这些命令包括 `F BIOGET_VSCREENINFO`（获得可变的屏幕参数）、`F BIOPUT_VSCREENINFO`（设置可变的屏幕参数）、`F BIOGET_FSCREENINFO`（获得固定的屏幕参数设置，注意，固定的屏幕参数不能由用户设置）、`F BIOPUTCMAP`（设置颜色表）、`F BIOGETCMAP`（获得颜色表）等。代码清单 18.14 所示为帧缓冲设备 `ioctl()` 函数的源代码。

代码清单 18.14 帧缓冲设备驱动的 `ioctl` 函数

```

1 long do_fb_ioctl(struct fb_info *info, unsigned int cmd,
2     unsigned long arg)
3 {
4     struct fb_ops *fb;
5     struct fb_var_screeninfo var;
6     struct fb_fix_screeninfo fix;
7     struct fb_con2fbmap con2fb;
8     struct fb_cmap_user cmap;
9     struct fb_event event;

```



```
10 void __user *argp = (void __user *)arg;
11 long ret = 0;
12
13 fb = info->fbops;
14 if (!fb)
15     return -ENODEV;
16
17 switch (cmd) {
18 case FBIOGET_VSCREENINFO:
19     ret = copy_to_user(argp, &info->var,
20         sizeof(var)) ? -EFAULT : 0;
21     break;
22 case FBIOPUT_VSCREENINFO:
23     if (copy_from_user(&var, argp, sizeof(var))) {
24         ret = -EFAULT;
25         break;
26     }
27     acquire_console_sem();
28     info->flags |= FBINFO_MISC_USEREVENT;
29     ret = fb_set_var(info, &var);
30     info->flags &= ~FBINFO_MISC_USEREVENT;
31     release_console_sem();
32     if (ret == 0 && copy_to_user(argp, &var, sizeof(var)))
33         ret = -EFAULT;
34     break;
35 case FBIOGET_FSCREENINFO:
36     ret = copy_to_user(argp, &info->fix,
37         sizeof(fix)) ? -EFAULT : 0;
38     break;
39 case FBIOPUTCMAP:
40     if (copy_from_user(&cmap, argp, sizeof(cmap)))
41         ret = -EFAULT;
42     else
43         ret = fb_set_user_cmap(&cmap, info);
44     break;
45 case FBIOGETCMAP:
46     if (copy_from_user(&cmap, argp, sizeof(cmap)))
47         ret = -EFAULT;
48     else
49         ret = fb_cmap_to_user(&info->cmap, &cmap);
50     break;
51 ...
52 default:
53     if (fb->fb_ioctl == NULL)
54         ret = -ENOTTY;
55     else
56         ret = fb->fb_ioctl(info, cmd, arg);
57 }
58 return ret;
59 }
60
61 static long fb_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
62 __acquires(&info->lock)
63 __releases(&info->lock)
64 {
```

```

65 struct inode *inode = file->f_path.dentry->d_inode;
66 int fbidx = iminor(inode);
67 struct fb_info *info;
68 long ret;
69
70 info = registered_fb[fbidx];
71 mutex_lock(&info->lock);
72 ret = do_fb_ioctl(info, cmd, arg);
73 mutex_unlock(&info->lock);
74 return ret;
75 }

```

18.9 帧缓冲设备的用户空间访问

通过/dev/fbn，应用程序可进行的针对帧缓冲设备的操作主要有如下几种。

- 读/写 dev/fbn：相当于读/写屏幕缓冲区。例如用 `cp /dev/fb0 tmp` 命令可将当前屏幕的内容复制到一个文件中，而命令 `cp tmp > /dev/fb0` 则将图形文件 tmp 显示在屏幕上。
- 映射操作：对于帧缓冲设备，可通过 `mmap()` 映射操作将屏幕缓冲区的物理地址映射到用户空间的一段虚拟地址中，之后用户就可以通过读/写这段虚拟地址访问屏幕缓冲区，在屏幕上绘图了。而且若干个进程可以映射到同一个显示缓冲区。实际上，使用帧缓冲设备的应用程序都是通过映射操作来显示图形的。
- I/O 控制：对于帧缓冲设备，对设备文件的 `ioctl()` 操作可读取/设置显示设备及屏幕的参数，如分辨率、显示颜色数、屏幕大小等。

如图 18.6 所示，在应用程序中，操作/dev/fbn 的一般步骤如下。

- (1) 打开/dev/fbn 设备文件。
- (2) 用 `ioctl()` 操作取得当前显示屏幕的参数，如屏幕分辨率、每个像素点的比特数和偏移。根据屏幕参数可计算屏幕缓冲区的大小。
- (3) 将屏幕缓冲区映射到用户空间。
- (4) 映射后就可以直接读/写屏幕缓冲区，进行绘图和图片显示了。

代码清单 18.15 所示为一段用户空间访问帧缓冲设备显示缓冲区的范例，包含打开和关闭帧缓冲设备、得到和设置可变参数、得到固定参数、生成与 BPP 对应的帧缓冲数据及填充显示缓冲区。

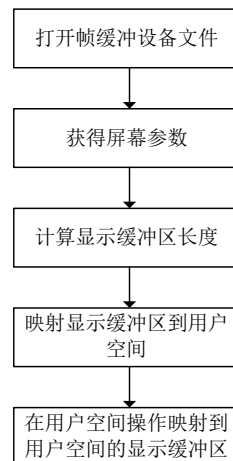


图 18.6 用户空间访问帧缓冲设备流程

代码清单 18.15 用户空间访问帧缓冲设备显示缓冲区范例

```

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <linux/fb.h>
#include <sys/mman.h>

```



```
int main()
{
    int fbfd = 0;
    struct fb_var_screeninfo vinfo;
    unsigned long screensize = 0;
    char *fbp = 0;
    int x = 0, y = 0;
    int i = 0;

    // Open the file for reading and writing
    fbfd = open("/dev/fb0", O_RDWR);
    if (!fbfd) {
        printf("Error: cannot open framebuffer device.\n");
        exit(1);
    }
    printf("The framebuffer device was opened successfully.\n");

    // Get variable screen information
    if (ioctl(fbfd, FBIOGET_VSCREENINFO, &vinfo)) {
        printf("Error reading variable information.\n");
        exit(1);
    }

    printf("%dx%d, %dbpp\n", vinfo.xres, vinfo.yres, vinfo.bits_per_pixel);
    if (vinfo.bits_per_pixel != 16) {
        printf("Error: not supported bits_per_pixel, it only supports 16 bit color\n");
        exit(1);
    }

    // Figure out the size of the screen in bytes
    screensize = vinfo.xres * vinfo.yres * 2;

    // Map the device to memory
    fbp = (char *)mmap(0, screensize, PROT_READ | PROT_WRITE, MAP_SHARED,
        fbfd, 0);
    if ((int)fbp == -1) {
        printf("Error: failed to map framebuffer device to memory.\n");
        exit(4);
    }
    printf("The framebuffer device was mapped to memory successfully.\n");

    // Draw 3 rect with graduated RED/GREEN/BLUE
    for (i = 0; i < 3; i++) {
        for (y = i * (vinfo.yres / 3); y < (i + 1) * (vinfo.yres / 3); y++) {
            for (x = 0; x < vinfo.xres; x++) {
                long location = x * 2 + y * vinfo.xres * 2;
                int r = 0, g = 0, b = 0;
                unsigned short rgb;

                if (i == 0)
                    r = ((x * 1.0) / vinfo.xres) * 32;
                if (i == 1)
                    g = ((x * 1.0) / vinfo.xres) * 64;
                if (i == 2)
                    b = ((x * 1.0) / vinfo.xres) * 32;
```



```

        rgb = (r << 11) | (g << 5) | b;
        *((unsigned short*)(fbp + location)) = rgb;
    }
}

munmap(fbp, screensize);
close(fbfd);
return 0;
}

```

上述程序位于虚拟机的/home/lihacker/develop/svn/ldd6410-read-only/tests/framebuffer 中，运行时会在屏幕上绘制 R/G/B 这 3 种颜色的由浅入深的变化情况。LDD6410 的文件系统中已经集成了这个 fb_test 程序，可以直接运行查看效果。

18.10 Linux 图形用户界面

18.10.1 Qt-X11/QtEmbedded/Qttopia

Qt 是 Trolltech (奇趣科技, 目前已被诺基亚收购) 公司所开发的一个跨平台 Framework 环境, 它采用类似 C++ 的语法, 在 Microsoft Windows、MacOS X、Linux、Solaris、HP-UX、Tru64 (Digital UNIX)、Irix、FreeBSD、BSD/OS、SCO、AIX 等平台上都可执行。

Trolltech 也针对嵌入式环境推出了 Qt/Embedded 产品。与桌面版本不同, Qt/Embedded 未采用 X Server 及 X Library 等角色, 而是直接使用帧缓冲作为底层图形接口 (如图 18.7 所示)。Qt/Embedded 提供了丰富的窗口小部件 (Widgets), 并且还支持窗口部件的定制, 因此它可以为用户提供漂亮的图形界面, 许多基于 Qt 的 X Window 程序可以非常方便地移植到 Qt/Embedded 版本上。

Qttopia 是建立在 Qt/Embedded 上的一种开放源代码窗口系统, 它与实际的产品相似, 专门针对 PDA、SmartPhone 这类运行嵌入式 Linux 的移动计算设备和手持设备而开发的。Trolltech 还发布了一款供应用开发人员使用的 Linux 手机 “Qttopia Greenphone”。

在宿主机上可通过 qvfb (虚拟帧缓冲) 来模拟帧缓冲。qvfb 是 X 窗口用来运行和测试 Qttopia 应用程序的系统程序, 它使用了共享存储区域 (虚拟的帧缓冲) 来模拟帧缓冲并且在一个窗口中模拟一个应用来显示帧缓冲, 允许我们在桌面及其上开发 Qt 嵌入式程序。

信号 (Signal) 和插槽 (Slot) 是 Qt 中一种用于对象间通信的调用机制, 不同于传统的函数回调方式, 信号和插槽是 Qt 中非常有特色的地方, 是 Qt 编程区别于其他编程的标志。信号和插槽不是标准 C++ 功能, C++ 编译器不能理解这些语句, 必须经过特殊的工具对象编辑器

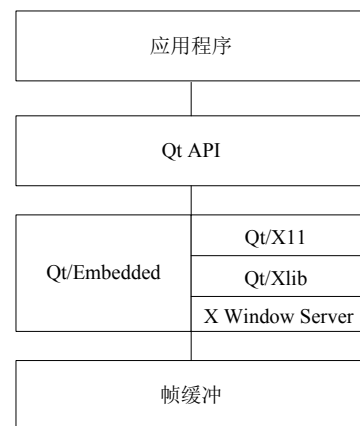


图 18.7 Qt/Embedded 与 Qt 的区别



MOC (Meta Object Compiler) 将源代码中创建信号和插槽的语句翻译成 C++ 编译器能够理解的代码。

Qt 的窗口在事件发生后激发信号。例如, 一个按钮被点击时会激发一个 “clicked” 信号。程序员通过建立一个函数 (称做一个插槽), 然后调用 `connect()` 函数把这个插槽和一个信号连接起来, 这样就完成了一个事件和响应代码的连接。信号与插槽机制并不要求类之间互相知道细节, 这样就可以相对容易地开发出代码可高重用的类。

例如, 如果一个退出按钮的 `clicked()` 信号被连接到了一个应用的退出函数 `quit()` 插槽。那么一个用户点击退出键将使应用程序终止运行, 完成上述连接过程的代码如下。

```
connect( button, SIGNAL(clicked()), qApp, SLOT(quit()) );
```

代码清单 18.16 的应用程序创建一个 `hello` 窗口, 该窗口显示一个动态字符串 “Hello, World”, 程序中添加了一个 `QTimer` 定时器实例, 以周期性刷新屏幕, 从而得到动画的效果。

代码清单 18.16 Qt/Embedded 应用程序范例

```
1  /*****
2  ** 以下是 hello.h 的代码
3  *****/
4  #ifndef HELLO_H
5      #define HELLO_H
6      #include <qvariant.h>
7      #include <qwidget.h>
8      class QVBoxLayout;
9      class QHBoxLayout;
10     class QGridLayout;
11     class Hello: public QWidget
12     {
13     public:
14         Hello(QWidget *parent = 0, const char *name = 0, WFlags fl = 0);
15         ~Hello();
16         //以下是手动添加的代码
17         signals: void clicked();
18     protected:
19         void mouseReleaseEvent(QMouseEvent*);
20         void paintEvent(QPaintEvent*);
21         private slots: void animate();
22     private:
23         QString t;
24         int b;
25     };
26 #endif // HELLO_H
27
28 /*****
29 ** 以下是 hello.cpp 源代码
30 *****/
31 #include "hello.h"
32 #include <qlayout.h>
33 #include <qvariant.h>
34 #include <qtooltip.h>
35 #include <qwhatsthis.h>
36 #include <qpushbutton.h>
37 #include <qtimer.h>
```

```

38 #include <qpainter.h>
39 #include <qpixmap.h>
40 /* 构造一个 Hello 窗口 */
41 Hello::Hello(QWidget *parent, const char *name, WFlags fl): QWidget(parent,
42     name, fl)
43 {
44     if (!name)
45         setName("Hello");
46     resize(240, 320);
47     setMinimumSize(QSize(240, 320));
48     setMaximumSize(QSize(240, 320));
49     setSizeIncrement(QSize(240, 320));
50     setBaseSize(QSize(240, 320));
51     QPalette pal;
52     QColorGroup cg;
53     cg.setColor(QColorGroup::Foreground, black);
54     cg.setColor(QColorGroup::Button, QColor(192, 192, 192));
55     cg.setColor(QColorGroup::Light, white);
56     cg.setColor(QColorGroup::Midlight, QColor(223, 223, 223));
57     cg.setColor(QColorGroup::Dark, QColor(96, 96, 96));
58     cg.setColor(QColorGroup::Mid, QColor(128, 128, 128));
59     cg.setColor(QColorGroup::Text, black);
60     cg.setColor(QColorGroup::BrightText, white);
61     cg.setColor(QColorGroup::ButtonText, black);
62     cg.setColor(QColorGroup::Base, white);
63     cg.setColor(QColorGroup::Background, white);
64     cg.setColor(QColorGroup::Shadow, black);
65     cg.setColor(QColorGroup::Highlight, black);
66     cg.setColor(QColorGroup::HighlightedText, white);
67     pal.setActive(cg);
68     cg.setColor(QColorGroup::Foreground, black);
69     cg.setColor(QColorGroup::Button, QColor(192, 192, 192));
70     cg.setColor(QColorGroup::Light, white);
71     cg.setColor(QColorGroup::Midlight, QColor(220, 220, 220));
72     cg.setColor(QColorGroup::Dark, QColor(96, 96, 96));
73     cg.setColor(QColorGroup::Mid, QColor(128, 128, 128));
74     cg.setColor(QColorGroup::Text, black);
75     cg.setColor(QColorGroup::BrightText, white);
76     cg.setColor(QColorGroup::ButtonText, black);
77     cg.setColor(QColorGroup::Base, white);
78     cg.setColor(QColorGroup::Background, white);
79     cg.setColor(QColorGroup::Shadow, black);
80     cg.setColor(QColorGroup::Highlight, black);
81     cg.setColor(QColorGroup::HighlightedText, white);
82     pal.setInactive(cg);
83     cg.setColor(QColorGroup::Foreground, QColor(128, 128, 128));
84     cg.setColor(QColorGroup::Button, QColor(192, 192, 192));
85     cg.setColor(QColorGroup::Light, white);
86     cg.setColor(QColorGroup::Midlight, QColor(220, 220, 220));
87     cg.setColor(QColorGroup::Dark, QColor(96, 96, 96));
88     cg.setColor(QColorGroup::Mid, QColor(128, 128, 128));
89     cg.setColor(QColorGroup::Text, black);
90     cg.setColor(QColorGroup::BrightText, white);
91     cg.setColor(QColorGroup::ButtonText, QColor(128, 128, 128));
92     cg.setColor(QColorGroup::Base, white);

```



```
93     cg.setColor(QColorGroup::Background, white);
94     cg.setColor(QColorGroup::Shadow, black);
95     cg.setColor(QColorGroup::Highlight, black);
96     cg.setColor(QColorGroup::HighlightedText, white);
97     pal.setDisabled(cg);
98     setPalette(pal);
99     QFont f(font());
100    f.setFamily("adobe-helvetica");
101    f.setPointSize(29);
102    f.setBold(TRUE);
103    setFont(f);
104    setCaption(tr(""));
105    t = "Hello,World";
106    b = 0;
107    QTimer *timer = new QTimer(this); //创建定时器
108    connect(timer, SIGNAL(timeout()), SLOT(animate())); //连接信号和插槽
109    timer->start(40);
110 }
111
112 /* 销毁对象, 释放任何被分配的资源 */
113 Hello::~~Hello(){}
114
115 /* 每次定时器到期后调用插槽 */
116 void Hello::animate()
117 {
118     b = (b + 1) &15;
119     repaint(FALSE); //重绘
120 }
121
122 /* 处理 hello 窗口的鼠标按钮释放事件 */
123 void Hello::mouseReleaseEvent(QMouseEvent *e)
124 {
125     if (rect().contains(e->pos()))
126         emit clicked(); //激活 clicked() 信号
127 }
128
129 /* 处理 hello 窗口的重绘事件 */
130 void Hello::paintEvent(QPaintEvent*)
131 {
132     static int sin_tbl[16] =
133     {
134         0, 38, 71, 92, 100, 92, 71, 38, 0, -38, -71, -92, -100, -92,
135         -71, -38
136     };
137     if (t.isEmpty())
138         return ;
139     // 1: 计算尺寸、位置
140     QFontMetrics fm = fontMetrics();
141     int w = fm.width(t) + 20;
142     int h = fm.height() *2;
143     int pmx = width() / 2-w / 2;
144     int pmy = height() / 2-h / 2;
145     // 2: 创建 pixmap, 用窗口背景填充它
146     QPixmap pm(w, h);
147     pm.fill(this, pmx, pmy);
```

```

148 // 3: 绘制 pixmap
149 QPainter p;
150 int x = 10;
151 int y = h / 2 + fm.descent();
152 int i = 0;
153 p.begin(&pm);
154 p.setFont(font());
155 while (!t[i].isNull())
156 {
157     int il6 = (b + i) % 15;
158     p.setPen(QColor((15-il6) * 16, 255, 255, QColor::Hsv));
159     p.drawText(x, y - sin_tbl[il6] * h / 800, t.mid(i, 1), 1);
160     x += fm.width(t[i]);
161     i++;
162 }
163 p.end();
164 // 4: 复制 pixmap 到 Hello 窗口
165 bitBlt(this, pmx, pmy, &pm);
166 }
167
168 /*****
169 ** 以下是 main.cpp 的源代码
170 *****/
171 #include "hello.h"
172 #include <qapplication.h>
173 /*
174 The program starts here. It parses the command line and builds a message
175 string to be displayed by the Hello widget.
176 */
177 #define QT_NO_WIZARD
178 int main(int argc, char **argv)
179 {
180     QApplication a(argc, argv);
181     Hello dlg;
182     QObject::connect(&dlg, SIGNAL(clicked()), &a, SLOT(quit()));
183     a.setMainWidget(&dlg);
184     dlg.show();
185     return a.exec();
186 }

```

18.10.2 Microwindows/Nano-X

Microwindows 是嵌入式系统中广为使用的一种图形用户接口，其官方网站是 <http://www.microwindows.org>，Microwindows 完全支持 Linux 的帧缓冲技术。这个项目的早期目标是在嵌入式 Linux 平台上提供和普通个人电脑上类似的图形用户界面。

Microwindows 起源于 NanoGUI 项目，早期 Microwindows 有两个版本，一个版本包含了一组和微软的 WIN32 图形用户接口相似的 API，这个版本就是 Microwindows 版本；另外一个版本是基于 X-Windows 的一组 Xlib 风格的 API 函数库，这个版本允许 X11 的二进制代码直接在 Micro Windows 的 Nanx-X 服务器上运行，称之为 Nano-X。

如图 18.8 所示，Microwindows 采用分层设计方法。在最底层，屏幕、鼠标/触摸屏以及键盘驱动程序提供了对物理设备访问的能力。在中间层，实现了一个可移植的图形引擎，支持行绘制、



区域填充、剪切以及颜色模型等。在上层, 实现多种 API 以适应不同的应用环境。

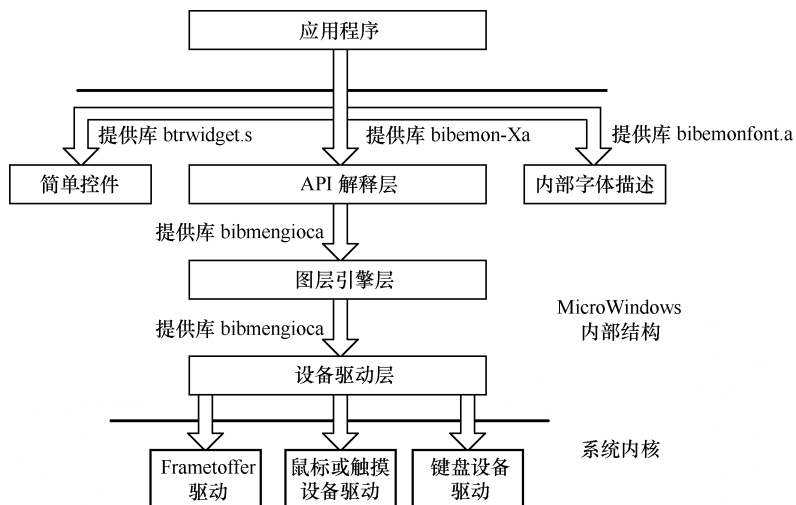


图 18.8 MicroWindows 的层次结构

代码清单 18.17 所示为一个简单的 MicroWindows 应用程序, 它基于 Nano-X API 编写, 创建一个窗口并显示“Hello World”, 如图 18.9 所示。

代码清单 18.17 Nano-X 应用程序范例

```
1 #include <stdio.h>
2 #include <microwin/nano-X.h>
3
4 GR_WINDOW_ID wid;
5 GR_GC_ID gc;
6
7 void event_handler(GR_EVENT *event);
8
9 int main(void)
10 {
11     if (GrOpen() < 0)
12     {
13         fprintf(stderr, "GrOpen failed");
14         exit(1);
15     }
16
17     gc = GrNewGC();
18     GrSetGCForeground(gc, 0xFF0000);
19     //创建窗口
20     wid = GrNewWindowEx(GR_WM_PROPS_APPFRAME |
21                        GR_WM_PROPS_CAPTION |
22                        GR_WM_PROPS_CLOSEBOX,
23                        "jollen.org", GR_ROOT_WINDOW_ID, 0, 0, 200,
24                        200, 0xFFFFFFFF);
25     //选择事件
26     GrSelectEvents(wid, GR_EVENT_MASK_CLOSE_REQ|GR_EVENT_MASK_EXPOSURE);
27
28     GrMapWindow(wid);
```

```

29 GrMainLoop(event_handler); //挂接事件处理函数
30 }
31
32 void event_handler(GR_EVENT *event)
33 {
34     switch (event->type)
35     {
36         case GR_EVENT_TYPE_EXPOSURE: //显示文本
37             GrText(wid, gc, 50, 50, "Hello World", - 1, GR_TFASCII);
38             break;
39         case GR_EVENT_TYPE_CLOSE_REQ: //关闭
40             GrClose();
41             exit(0);
42         default:
43             break;
44     }
45 }

```



图 18.9 Microwindows 使用范例

18.10.3 MiniGUI

MiniGUI 是由北京飞漫软件技术有限公司开发的面向实时嵌入式系统的轻量级图形用户界面支持系统，1999 年初遵循 GPL 条款发布第一个版本以来，已广泛应用于手持信息终端、机顶盒、工业控制系统及工业仪表、彩票机、金融终端等产品和领域。目前，MiniGUI 已成为跨操作系统的图形用户界面支持系统，可在 Linux/μClinux、eCos、μC/OS-II、VxWorks 等操作系统上运行，已验证的硬件平台包括 Intel x86、ARM、PowerPC、MIPS 和 M68K (DragonBall/ColdFire) 等。

如图 18.10 所示，基于 MiniGUI 的应用程序一般通过 ANSI C 库以及 MiniGUI 自身提供的 API 来实现自己的功能；MiniGUI 中的可移植层可将特定操作系统及底层硬件的细节隐藏起来，而上层应用程序则无须关心底层的硬件平台输出和输入设备。

为了适合不同的操作系统环境，MiniGUI 可配置成以下 3 种运行模式。



1. MiniGUI-Threads

运行在 MiniGUI-Threads 上的程序可以在不同的线程中建立多个窗口,但所有的窗口在一个进程或者地址空间中运行。这种运行模式非常适合于大多数传统意义上的嵌入式操作系统,比如 μ C/OS-II、eCos、VxWorks、pSOS 等。当然,在 Linux 和 μ CLinux 上,MiniGUI 也能以 MiniGUI-Threads 模式运行。

2. MiniGUI-Processes

和 MiniGUI-Threads 相反,MiniGUI-Processes 上的每个程序是单独的进程,每个进程也可以建立多个窗口,并且实现了多进程窗口系统。MiniGUI-Processes 适合于具有完整 UNIX 特性的嵌入式操作系统,比如嵌入式 Linux。

3. MiniGUI-Standalone

在这种运行模式下,MiniGUI 可以以独立进程的方式运行,既不需要多线程也不需要多进程的支持,这种运行模式适合功能单一的应用场合。

MiniGUI 下的通信是一种类似于 Win32 的消息机制,如果有 Win32 图形用户界面程序的编程基础,编写 MiniGUI 程序将没有门槛。代码清单 18.18 所示为一个完整的 MiniGUI 应用程序,该程序在屏幕上创建一个大小为 240×180 的应用程序窗口,并在窗口客户区的中部显示“Hello world!”,如图 18.11 所示。

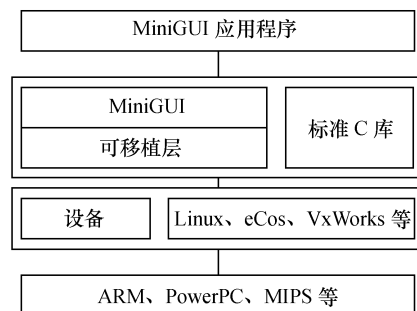


图 18.10 MiniGUI 和嵌入式操作系统的关系

代码清单 18.18 MiniGUI 应用程序范例

```
1 #include <stdio.h>
2 #include <minigui/common.h>
3 #include <minigui/minigui.h>
4 #include <minigui/gdi.h>
5 #include <minigui/window.h>
6 static int HelloWinProc ( HWND hWnd, int message, WPARAM wParam, LPARAM lParam )
7 {
8     HDC hdc;
9     switch (message)
10    {
11        case MSG_PAINT:
12            hdc = BeginPaint(hWnd);
13            TextOut(hdc, 60, 60, "Hello world!"); //输出文本
14            EndPaint(hWnd, hdc);
15            return 0;
16        case MSG_CLOSE:
17            DestroyMainWindow(hWnd); //破坏窗口
18            PostQuitMessage(hWnd); //释放退出消息
19            return 0;
20    }
21    return DefaultMainWinProc(hWnd, message, wParam, lParam);
22 }
23
24 int MiniGUIMain(int argc, const char *argv[])
25 {
26     MSG Msg;
27     HWND hMainWnd; //主窗口句柄
28     MAINWINCREATE CreateInfo;
```



```

29  #ifdef _MGRM_PROCESSES
30      JoinLayer(NAME_DEF_LAYER, "helloworld", 0, 0);
31  #endif
32  CreateInfo.dwStyle = WS_VISIBLE | WS_BORDER | WS_CAPTION;
33  CreateInfo.dwExStyle = WS_EX_NONE;
34  CreateInfo.spCaption = "HelloWorld";
35  CreateInfo.hMenu = 0;
36  CreateInfo.hCursor = GetSystemCursor(0);
37  CreateInfo.hIcon = 0;
38  CreateInfo.MainWindowProc = HelloWinProc; //主窗口消息处理程序
39  CreateInfo.lx = 0;
40  CreateInfo.ty = 0;
41  CreateInfo.rx = 240; //水平尺寸
42  CreateInfo.by = 180; //垂直尺寸
43  CreateInfo.iBkColor = COLOR_lightwhite;
44  CreateInfo.dwAddData = 0;
45  CreateInfo.hHosting = HWND_DESKTOP;
46  hMainWnd = CreateMainWindow(&CreateInfo); //创建主窗口
47  if (hMainWnd == HWND_INVALID)
48      return -1;
49  ShowWindow(hMainWnd, SW_SHOWNORMAL);
50  while (GetMessage(&Msg, hMainWnd))
51  {
52      TranslateMessage(&Msg); //消息译码
53      DispatchMessage(&Msg); //派送消息
54  }
55  MainWindowThreadCleanup(hMainWnd);
56  return 0;
57 }

```



图 18.11 MiniGUI 使用范例

18.10.4 Android

Android 是 google 推出的一个移动设备平台，其软件层次结构包括了一个操作系统（OS），中间件（MiddleWare）和应用程序（Application）。根据 Android 的软件框图，其软件层次结构自下而上分为以下几个层次。

- (1) 操作系统层（OS）。
- (2) 各种库（Libraries）和 Android 运行时（RunTime）。



(3) 应用程序框架 (Application Framework)。

(4) 应用程序 (Applications)。

Android 的应用程序主要涉及用户界面, 通常以 Java 程序编写。Android 本身提供了主屏幕 (Home)、联系人、电话、浏览器等众多的核心应用。同时应用程序的开发者还可以使用应用程序框架层的 API 实现自己的程序。目前 Android 的应用开发非常热门, 已有大量文档和书籍讲解, 本书不再赘述。

LDD6410 整合了 Android 1.6, LDD6410 的 Android 本身作为 Linux 文件系统的一部分进行管理。在系统启动后, 运行根目录下的 android, 系统将进入 Android (如图 18.12 所示)。LDD6410 Android 支持按键、触摸屏和鼠标操作, 显示设备可以是 LCD(480*272)和 VGA(1024*768@ 60Hz)。

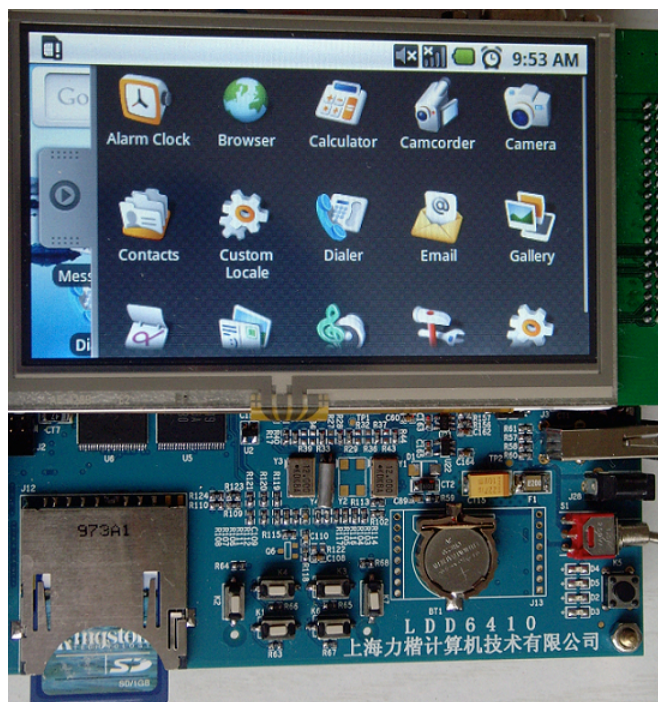


图 18.12 LDD6410 Android 界面

在本书配套光盘中提供了 Android 1.6 SDK android-sdk-linux_x86-1.6_r1.tgz, 使用该 SDK 就可以生成支持 LDD6410 的 Android, 其步骤是:

(1) 在 PC 上创建 LDD6410 虚拟机。

```
lihacker@lihacker - laptop: ~ /develop/LDD6410/android - sdk - linux_x86 - 1.6_r1/
tools$ ./android create avd -n LDD6410 -t 2
Android 1.6 is a basic Android platform.
Do you wish to create a custom hardware profile [no]y
Device ram size: The amount of physical RAM on the device, in megabytes. hw.ramSize
[96]:128
Touch-screen support: Whether there is a touch screen or not on the device.
hw.touchScreen [yes]:
Track-ball support: Whether there is a trackball on the device. hw.trackBall [yes]:n
Keyboard support: Whether the device has a QWERTY keyboard.
hw.keyboard [yes]:n
DPad support: Whether the device has DPad keys hw.dPad [yes]:y
GSM modem support: Whether there is a GSM modem in the device.
```

```

hw.gsmModem [yes]:n
Camera support: Whether the device has a camera. hw.camera [no]:n
Maximum horizontal camera pixels
hw.camera.maxHorizontalPixels [640]: Maximum vertical camera pixels hw.camera.
maxVerticalPixels [480]:
GPS support: Whether there is a GPS in the device.
hw.gps [yes]:n
Battery support: Whether the device can run on a battery. hw.battery [yes]:n
Accelerometer: Whether there is an accelerometer in the device.
hw.accelerometer [yes]:n
Audio recording support: Whether the device can record audio hw.audioInput [yes]:
Audio playback support: Whether the device can play audio
hw.audioOutput [yes]:
SD Card support: Whether the device supports insertion/removal of virtual SD Cards.
hw.sdCard [yes]:
Cache partition support: Whether we use a /cache partition on the device.
disk.cachePartition [yes]:n
Cache partition size disk.cachePartition.size [66MB]:
Abstracted LCD density: Must be one of 120, 160 or 240. A value used to roughly describe
the density of the LCD screen for automatic resource/asset selection.
hw.lcd.density [160]:
Created AVD 'LDD6410' based on Android 1.6, with the following hardware config:
hw.gps=no hw.dPad=yes hw.accelerometer=no hw.lcd.density=160 disk.cachePartition=no
hw.keyboard=no hw.trackBall=no hw.ramSize=128 hw.gsmModem=no hw.camera=no hw.battery=no

```

(2) 在主机上创建一个 sd card 的 image。

```
sudo ./mksdcard 128M sdcard.img
```

(3) 在主机上启动 Android 模拟器，运行如下命令启动 LDD6410 虚拟机，如图 18.3 所示：

```
sudo ./emulator -sdcard ./sdcard.img @LDD6410
```

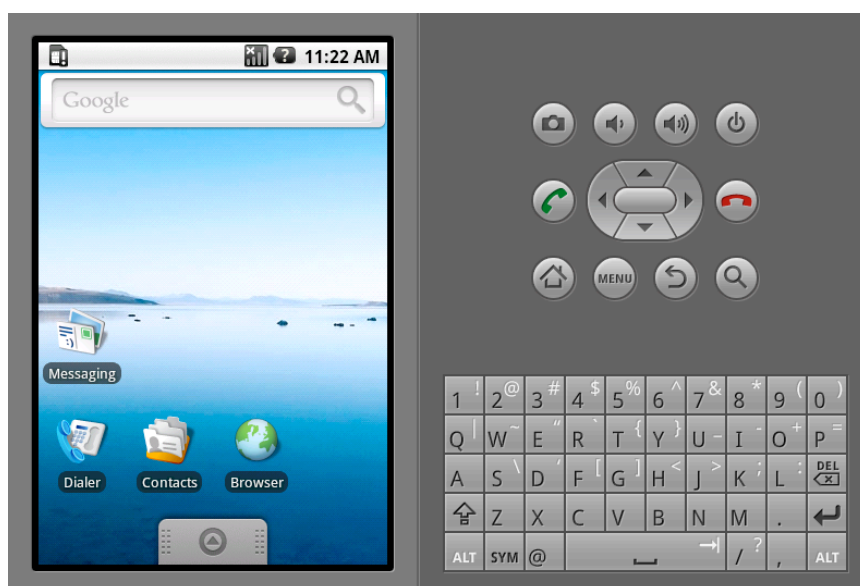


图 18.13 Android 虚拟机界面

启动 adb shell 连接 LDD6410 虚拟机，查看模拟器目标机上文件系统的挂载情况：

```
# mount
rootfs / rootfs ro 0 0
```



```
tmpfs /dev tmpfs rw,mode=755 0 0
devpts /dev/pts devpts rw,mode=600 0 0
proc /proc proc rw 0 0
sysfs /sys sysfs rw 0 0
tmpfs /sqlite_stmt_journals tmpfs rw,size=4096k 0 0
/dev/block/mtdblock0 /system yaffs2 ro 0 0
/dev/block/mtdblock1 /data yaffs2 rw,nosuid,nodev 0 0
/dev/block/vold/179:0 /sdcard vfat rw,dirsync,nosuid,nodev,noexec,uid=1000,gid=1015,fmask=
0702,dmask=0702,allow_utmime=0020,c odepag=cp437,ioccharset=iso8859-1,shortname=mixed,utf8 0 0
```

(4) 提取 Android 1.6。

把 busybox 放入模拟器目标机文件系统中：

```
./adb push ~/develop/svn/ldd6410/utills/busybox-1.15.1/_install/bin/busybox /data
```

我们现在把/system、/data、/sbin 目录以及根目录下的 init、init.rc 等都放入 sdcard 的 image 中：

```
# /data/busybox tar cvf /sdcard/android.tar /data /system /sbin /sqlite_stmt_journals
/init.rc
/init.goldfish.rc /init
```

进入/sdcard 目录看看得到的压缩文件：

```
# cd /sdcard
# ls -l
----rwxr-x system sdcard_rw 78487552 2010-01-30 12:37 android.tar
```

在主机上以 loop 方式 mount sdcard 的 image，并将里面的文件放到 LDD6410 目标电路板的根文件系统即可。

18.11 实例：S3C6410 LCD 设备驱动

S3C6410 内部集成了显示控制器，它可以将局部总线上来自后处理器（POST Processor）的图像数据和系统内存中的视频缓冲传输到外部的 LCD 接口上。图 18.14 给出了 S3C6410 显示控制器的框图，其 LCD 接口支持 4 种模式：传统的 RGB 接口、I80 接口、NTSC/PAL 标准电视接口和 IT-R BT. 601 接口。显示控制器针对视频数据的端口包括 RGB_VD[23:0]、SYS_VD[17:0]和 TV_OUT。

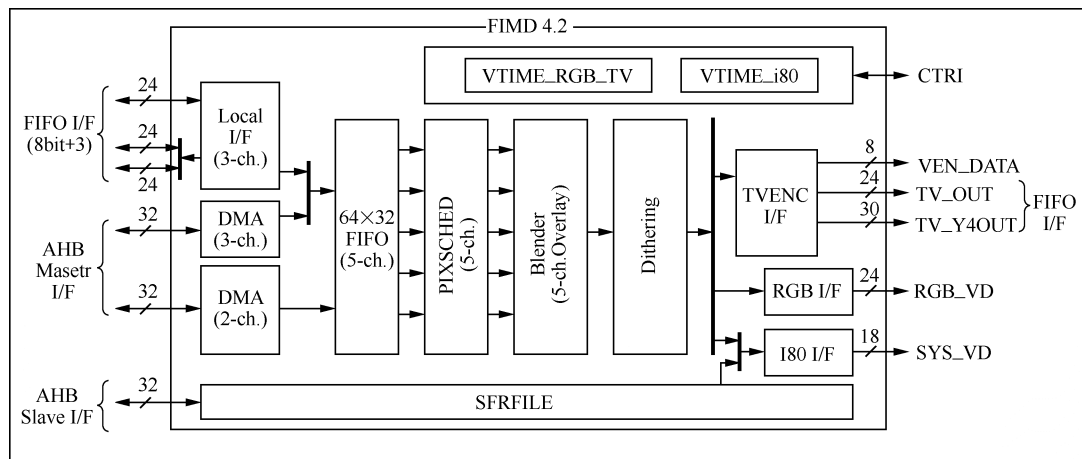


图 18.14 S3C6410 显示控制器逻辑结构

LDD6410 开发板上可连接外部 VGA 显示器和东华、群创等 LCD，VGA 部分透过 ADV7123 芯片进行数字信号向模拟信号的转换，连接 LCD 则无需此转换过程，图 18.15 所示 LDD6410 开发板上 S3C6410 芯片外围的 LCD 接口信号。

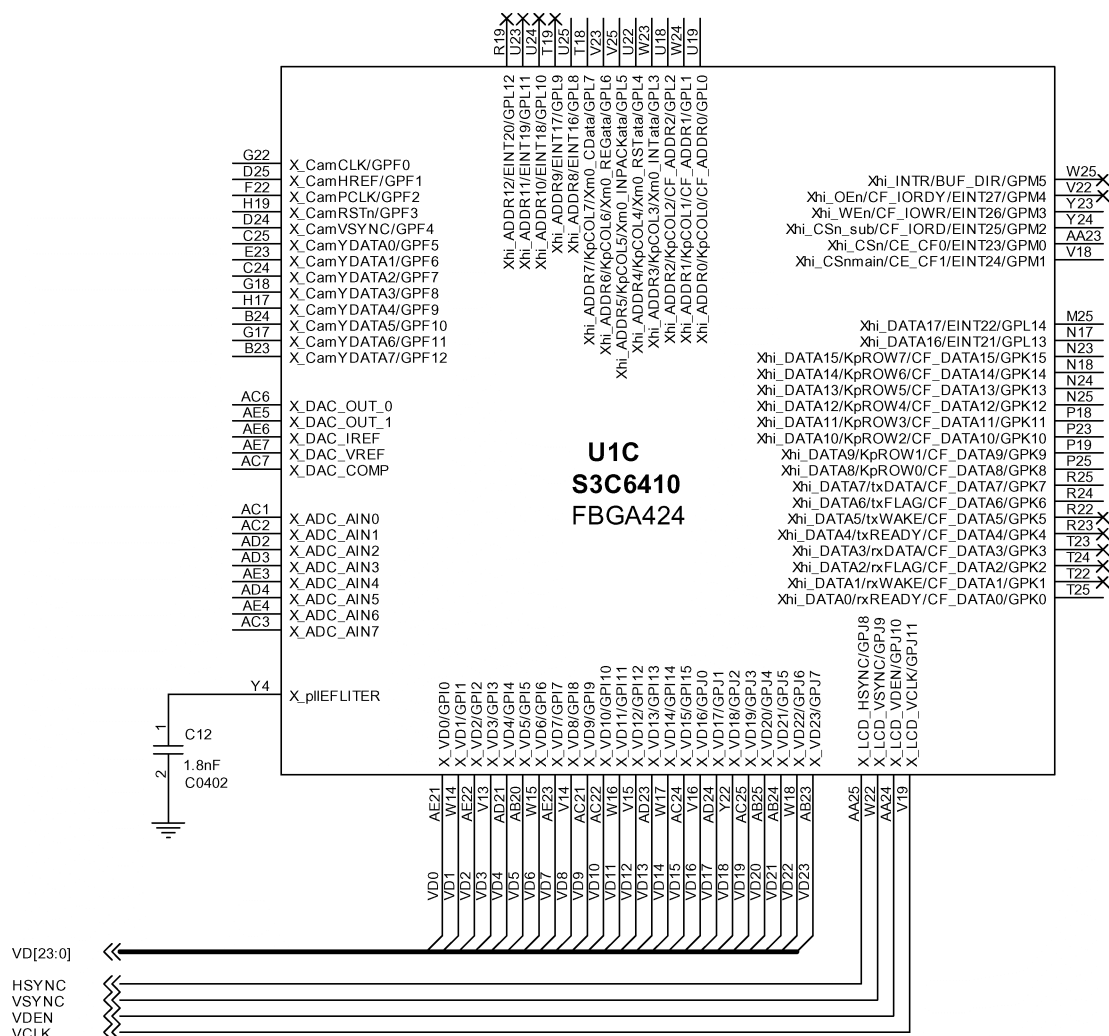


图 18.15 S3C6410 芯片外围的 LCD 接口信号

在本书配套光盘 Linux 2.6.28 内核源代码的 drivers/video/samsung 目录下，包含了 S3C6410 的 LCD 驱动。核心的工作由 s3cfb.c、s3cfb_fimd4x.c 等文件进行了实现，这些文件实现的主体内容是实现了 framebuffer 设备的注册、注销以及其中的 fb_ops 的功能函数，代码清单 18.19 抽取了 s3cfb.c 的框架。

代码清单 18.19 S3C6410 LCD 控制器驱动

```
1 struct fb_ops s3cfb_ops = {
2     .owner      = THIS_MODULE,
3     .fb_check_var = s3cfb_check_var,
```



```
4 .fb_set_par      = s3cfb_set_par,
5 .fb_blank       = s3cfb_blank,
6 .fb_pan_display= s3cfb_pan_display,
7 .fb_setcolreg   = s3cfb_setcolreg,
8 .fb_fillrect    = cfb_fillrect,
9 .fb_copyarea    = cfb_copyarea,
10 .fb_imageblit   = cfb_imageblit,
11 #ifdef CONFIG_FRAMEBUFFER_CONSOLE
12 .fb_cursor      = soft_cursor,
13 #endif
14 .fb_ioctl       = s3cfb_ioctl,
15 };
16
17 static void s3cfb_init_fbinfo(s3cfb_info_t *finfo, char *drv_name, int index)
18 {
19 ...
20 finfo->fb.fbops = &s3cfb_ops;
21 ..
22 }
23
24 static int __init s3cfb_probe(struct platform_device *pdev)
25 {
26 ...
27 fbinfo = framebuffer_alloc(sizeof(s3cfb_info_t), &pdev->dev);
28 ...
29 s3cfb_init_fbinfo(&s3cfb_info[index], driver_name, index);
30 ...
31 ret = s3cfb_init_registers(&s3cfb_info[index]);
32 ret = s3cfb_check_var(&s3cfb_info[index].fb.var, &s3cfb_info[index].fb);
33 ...
34 ret = register_framebuffer(&s3cfb_info[index].fb);
35 ...
36 }
37
38 static int s3cfb_remove(struct platform_device *pdev)
39 {
40 ...
41 unregister_framebuffer(&info[index].fb);
42
43 return 0;
44 }
45
46 static struct platform_driver s3cfb_driver = {
47 .probe      = s3cfb_probe,
48 .remove     = s3cfb_remove,
49 .suspend    = s3cfb_suspend,
50 .resume     = s3cfb_resume,
51 .driver     = {
52 .name       = "s3c-lcd",
53 .owner      = THIS_MODULE,
54 },
55 };
56
57 int __devinit s3cfb_init(void)
58 {
```

```

59 return platform_driver_register(&s3cfb_driver);
60 }
61 static void __exit s3cfb_cleanup(void)
62 {
63     platform_driver_unregister(&s3cfb_driver);
64 }
65
66 module_init(s3cfb_init);
67 module_exit(s3cfb_cleanup);

```

由于通用的文件实现了核心的工作，对于具体的 LCD 面板和 VGA 而言，我们只需要进行定时和硬件参数的配置了。LDD6410 开发板的东华 4.3 寸 LCD 面板（分辨率为 480*272）的配置文件位于 `drivers/video/samsung/s3cfb_wanxin.c`，其代码如清单 18.20。

代码清单 18.20 LDD6410 外接东华 4.3 寸 LCD 面板配置

```

1  #define S3CFB_HFP      2  /* front porch */
2  #define S3CFB_HSW      41 /* hsync width */
3  #define S3CFB_HBP      2  /* back porch */
4
5  #define S3CFB_VFP      2  /* front porch */
6  #define S3CFB_VSW      10 /* vsync width */
7  #define S3CFB_VBP      2  /* back porch */
8
9  #define S3CFB_HRES      480 /* horizon pixel x resolution */
10 #define S3CFB_VRES      272 /* line cnt y resolution */
11
12 #define S3CFB_HRES_VIRTUAL 480 /* horizon pixel x resolution */
13 #define S3CFB_VRES_VIRTUAL (272*2) /* line cnt y resolution */
14
15 #define S3CFB_HRES_OSD      480 /* horizon pixel x resolution */
16 #define S3CFB_VRES_OSD      272 /* line cnt y resolution */
17
18 #define S3CFB_VFRAME_FREQ      60 /* frame rate freq */
19
20 #define S3CFB_PIXEL_CLOCK (S3CFB_VFRAME_FREQ * (S3CFB_HFP + S3CFB_HSW + S3CFB_HBP \
21     + S3CFB_HRES) * (S3CFB_VFP + S3CFB_VSW + S3CFB_VBP + S3CFB_VRES))
22
23 static void s3cfb_set_fimd_info(void)
24 {
25     ...
26 }
27
28 int s3cfb_wanxin_set_gpio(void)
29 {
30     ...
31     return 0;
32 }
33
34
35 void s3cfb_init_hw(void)
36 {
37     printk(KERN_INFO "WANXIN LCD will be initialized\n");
38
39     s3cfb_set_fimd_info();

```



```
40 s3cfb_wanxin_set_gpio();  
41 }
```

S3CFB_HFP、S3CFB_HSW、S3CFB_HBP、S3CFB_VFP、S3CFB_VSW、S3CFB_VBP 等宏的值都直接取材于该款 LCD 面板的数据手册中的 7.4.2 节，如图 18.16 所示。

Parameter	Symbol	Min.	Typ.	Max.	Unit
Clock cycle	f _{CLK}	-	9	15	MHz
Hsync cycle	1/th	-	17.14	-	KHz
Vsync cycle	1/tv	-	59.94	-	Hz
Horizontal Signal					
Horizontal cycle	th ⁽¹⁾	-	525	-	CLK
Horizontal display period	thd	-	480	-	CLK
Horizontal front porch	thf	2	-	-	CLK
Horizontal pulse width	thp	2	41	-	CLK
Horizontal back porch	thb	2	2	-	CLK
Vertical Signal					
Vertical cycle	tv	-	286	-	H
Vertical display period	tvd	-	272	-	H
Vertical front porch	tvf	1	2	-	H
Vertical pulse width	tvp	1	10	-	H
Vertical back porch	tvb	1	2	-	H

Note:
(1) thd=480CLK, thf=2CLK, thp=41CLK, thb=2CLK, thf + thp + thb > 44CLK. (CLK=1/ fCLK, H=th)

图 18.16 LDD6410 外接东华 4.3 寸 LCD 面板的定时表

LDD6410 开发板外接 VGA 显示器的配置文件位于 `drivers/video/samsung/ s3cfb_vga.c`，其中的 S3CFB_HFP、S3CFB_HSW、S3CFB_HBP、S3CFB_VFP、S3CFB_VSW、S3CFB_VBP 等宏的取值则直接来源于 VESA（视频电子标准协会）工业标准。

光盘中附带的 LDD6410 工程源代码下的 `tests/framebuffer/fb_test.c` 文件实现了在电路板的 4.3 寸 LCD 显示器和 1024*768 分辨率 VGA 显示器上绘制色彩渐变的 R、G、B 三色矩形框的源代码，可以作为进一步学习 framebuffer 用户空间编程的实例。

18.12 总结

帧缓冲设备是一种典型的字符设备，它统一了显存，将显示缓冲区直接映射到用户空间。帧缓冲设备驱动 `file_operations` 中 `VFS` 接口函数由 `fbmem.c` 文件统一实现。这样，驱动工程师的工作重点将是实现针对特定设备 `fb_info` 中的 `fb_ops` 的成员函数，另外，理解并能灵活地修改 `fb_info` 中的 `var` 和 `fix` 参数非常关键。`fb_info` 中的 `var` 参数直接和 LCD 控制器的硬件设置以及 LCD 屏幕对应。

在用户空间，应用程序直接按照预先设置的 R、G、B 位数和偏移写经过 `mmap()` 映射后的显示缓冲区就可实现图形的显示，省去了内存从用户空间到内核空间的复制过程。

LINUX

第19章

Flash 设备驱动

Flash 在嵌入式系统中是必不可少的，它是 BootLoader、Linux 内核和文件系统的最佳载体。在 Linux 内核中，引入了 MTD 层为 NOR Flash 和 NAND Flash 设备提供统一的接口，从而使得 Flash 驱动的设计工作大为简化。

19.1 节讲解了 Linux Flash 驱动的结构，主要讲解了 MTD 系统的层次结构和接口。

19.2 节和 19.3 节分别讲解了 NOR Flash 和 NAND Flash 驱动的设计方法，给出了设计模板。

19.4 节和 19.5 节分别以 S3C6410 外围 NOR Flash 和 NAND Flash 为实例进一步讲解了 NOR Flash 和 NAND Flash 驱动的设计。

19.6 节讲解了如何在 Flash 上建立 cramfs、jffs/jffs2、yaffs/yaffs2 及 ubifs 文件系统。





19.1 Linux Flash 驱动结构

19.1.1 Linux MTD 系统层次

在 Linux 系统中, 提供了 MTD (Memory Technology Device, 内存技术设备) 系统来建立 Flash 针对 Linux 的统一、抽象的接口。MTD 将文件系统与底层的 Flash 存储器进行了隔离, 使 Flash 驱动工程师无须关心 Flash 作为字符设备和块设备与 Linux 内核的接口。

如图 19.1 所示, 在引入 MTD 后, Linux 系统中的 Flash 设备驱动及接口可分为 4 层, 从上到下依次是: 设备节点、MTD 设备层、MTD 原始设备层和硬件驱动层, 这 4 层的作用如下。

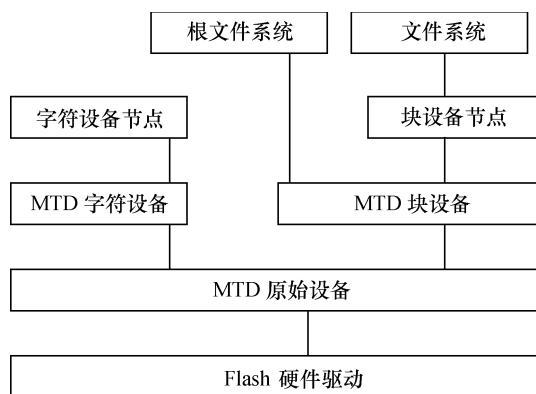


图 19.1 Linux MTD 系统

- 硬件驱动层: Flash 硬件驱动层负责 Flash 硬件设备的读、写、擦除, Linux MTD 设备的 NOR Flash 芯片驱动位于 `drivers/mtd/chips` 子目录下, NAND 型 Flash 的驱动程序则位于 `/drivers/mtd/nand` 子目录下。
- MTD 原始设备层: MTD 原始设备层由两部分组成, 一部分是 MTD 原始设备的通用代码, 另一部分是各个特定的 Flash 的数据, 例如分区。
- MTD 设备层: 基于 MTD 原始设备, Linux 系统可以定义出 MTD 的块设备 (主设备号 31) 和字符设备 (设备号 90), 构成 MTD 设备层。MTD 字符设备的定义在 `mtdchar.c` 中实现, 通过注册一系列 `file_operation` 函数 (`lseek`、`open`、`close`、`read`、`write`、`ioctl`) 可实现对 MTD 设备的读写和控制。MTD 块设备则是定义了一个描述 MTD 块设备的结构 `mtdblk_dev`, 并声明了一个名为 `mtdblks` 的指针数组, 这数组中的每一个 `mtdblk_dev` 和 `mtd_table` 中的每一个 `mtd_info` 一一对应。
- 设备节点: 通过 `mknod` 在 `/dev` 子目录下建立 MTD 字符设备节点 (主设备号为 90) 和 MTD 块设备节点 (主设备号为 31), 用户通过访问此设备节点即可访问 MTD 字符设备和块设备。

19.1.2 Linux MTD 系统接口

如图 19.2 所示, 在引入 MTD 后, 底层 Flash 驱动直接与 MTD 原始设备层交互, 利用其提供的接口注册设备和分区。

用于描述 MTD 原始设备的数据结构是 `mtd_info`, 这其中定义了大量关于 MTD 的数据和操作函数, 这个结构体的定义如代码清单 19.1 所示。`mtd_info` 是表示 MTD 原始设备的结构体, 每个分区也被认为是一个 `mtd_info`, 例如, 如果有两个 MTD 原始设备, 而每个上有 3 个分区, 在系统中就将共有 6 个 `mtd_info` 结构体, 这些 `mtd_info` 的指针被存放在名为 `mtd_table` 的数组里。

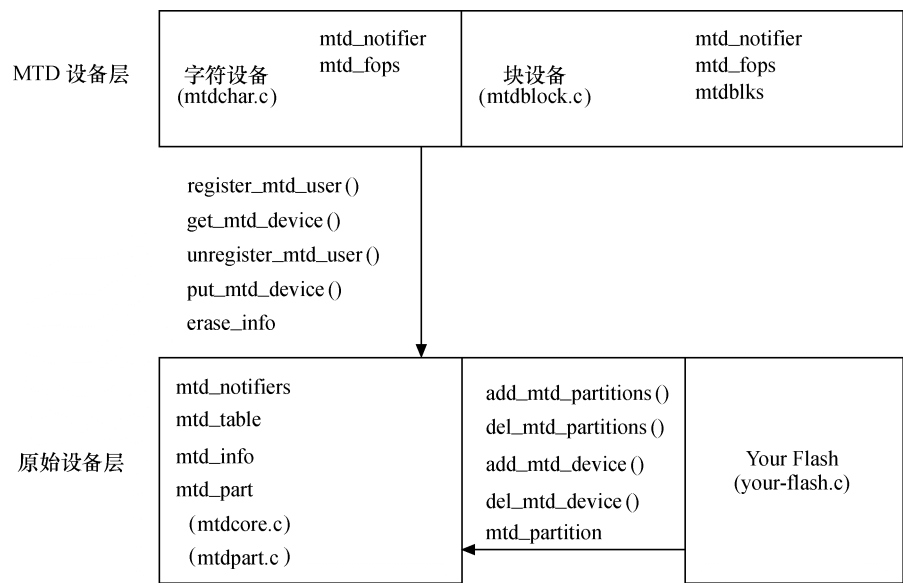


图 19.2 底层 Flash 驱动

代码清单 19.1 mtd_info 结构体

```
1 struct mtd_info {
2     u_char type; /* 内存技术的类型 */
3     u_int32_t flags; /*标志位*/
4     u_int32_t size; /*mtd 设备的大小*/
5     u_int32_t erasesize; /*主要的擦除块大小*/
6     u_int32_t writesize; /*最小的可写单元的字节数*/
7     u_int32_t oobsize; /* OOB 字节数*/
8     u_int32_t oobavail; /*可用的 OOB 字节数*/
9
10    char *name;
11    int index;
12    struct nand_ecclayout *ecclayout; /*ECC 布局结构体指针*/
13
14    /*不同的 erasesize 的区域*/
15    int numeraseregions; /*不同 erasesize 的区域的数目 (通常是 1)*/
16    struct mtd_erase_region_info *eraseregions;
17
18    u_int32_t bank_size;
19    struct module *module;
20    int(*erase)(struct mtd_info *mtd, struct erase_info *instr);
21
22    /*针对 eXecute-In-Place */
23    int (*point)(struct mtd_info *mtd, loff_t from, size_t len,
24        size_t *retlen, void **virt, resource_size_t *phys);
25
26    /* 如果 unpoint 为空, 不允许 XIP */
27    void (*unpoint)(struct mtd_info *mtd, loff_t from, size_t len);
28
29    int(*read)(struct mtd_info *mtd, loff_t from, size_t len, size_t *retlen,
30        u_char *buf); /*读 Flash */
```



```
31 int(*write)(struct mtd_info *mtd, loff_t to, size_t len, size_t *retlen,  
32     const u_char *buf); /*写 Flash */  
33 int (*panic_write) (struct mtd_info *mtd, loff_t to,  
34     size_t len, size_t *retlen, const u_char *buf); /*Kernel panic时继续写*/  
35 int (*read_oob) (struct mtd_info *mtd, loff_t  
36     from, struct mtd_oob_ops *ops); /*读 out-of-band */  
37 int (*write_oob) (struct mtd_info *mtd, loff_t to,  
38     struct mtd_oob_ops *ops); /*写 out-of-band */  
39 /* iovec-based 读写函数 */  
40 int (*writev) (struct mtd_info *mtd, const struct kvec  
41     *vecs, unsigned long count, loff_t to, size_t *retlen);  
42  
43 /* Sync */  
44 void(*sync)(struct mtd_info *mtd);  
45  
46 /* 设备锁 */  
47 int(*lock)(struct mtd_info *mtd, loff_t ofs, size_t len);  
48 int(*unlock)(struct mtd_info *mtd, loff_t ofs, size_t len);  
49 /* 电源管理函数*/  
50 int(*suspend)(struct mtd_info *mtd);  
51 void(*resume)(struct mtd_info *mtd);  
52  
53 /* 坏块管理函数*/  
54 int (*block_isbad) (struct mtd_info *mtd, loff_t ofs);  
55 int (*block_markbad) (struct mtd_info *mtd, loff_t ofs);  
56 ...  
57 void *priv; /*私有数据*/  
58 ...  
59 }
```

mtd_info 的 type 字段给出底层物理设备的类型, 包括 MTD_RAM、MTD_ROM、MTD_NORFLASH、MTD_NANDFLASH 等。

flags 字段标志可以是 MTD_WRITEABLE、MTD_BIT_WRITEABLE、MTD_NO_ERASE、MTD_POWERUP_LOCK 等的组合。针对 ROM 而言, 不具有上述任何属性, 因此 MTD_CAP_ROM 定义为 0; MTD_CAP_RAM 是 MTD_WRITEABLE、MTD_BIT_WRITEABLE、MTD_NO_ERASE 的组合; MTD_CAP_NORFLASH 是 MTD_WRITEABLE、MTD_BIT_WRITEABLE 的组合, 而对 MTD_CAP_NANDFLASH 则仅意味着 MTD_WRITEABLE。

mtd_info 中的 read()、write()、read_oob()、write_oob()、erase()是 MTD 设备驱动要实现的主要函数, 后面我们将看到, 在 NOR 和 NAND 的驱动代码中几乎看不到 mtd_info 的成员函数 (也即这些成员函数对于 Flash 芯片驱动是透明的), 这是因为 Linux 在 MTD 的下层实现了针对 NOR Flash 和 NAND Flash 的通用的 mtd_info 成员函数。

某些内存技术支持带外数据 (OOB), 例如, NAND Flash 每 512 字节就会有 16 个字节的“额外数据”, 用于存放纠错码或文件系统元数据。这是因为, 所有 Flash 器件都受“位翻转”现象的困扰, 而 NAND 发生的概率比 NOR 大, 因此 NAND 厂商推荐在使用 NAND 的时候最好要使用 ECC (Error Checking and Correcting), 汉明码是最简单的 ECC。mtd_info 的 ecclayout 类型即是描述 OOB 区域中 ECC 字节的布局情况。

Flash 驱动中使用如下两个函数注册和注销 MTD 设备:

```
int add_mtd_device(struct mtd_info *mtd);  
int del_mtd_device (struct mtd_info *mtd);
```

代码清单 19.2 所示的 `mtd_part` 结构体用于表示分区，其 `mtd_info` 结构体成员用于描述该分区，它会被加入到 `mtd_table`（定义为 `struct mtd_info *mtd_table[MAX_MTD_DEVICES]`）中，其大部分成员由其主分区 `mtd_part→master` 决定，各种函数也指向主分区的相应函数。

代码清单 19.2 `mtd_part` 结构体

```
1 struct mtd_part {
2     struct mtd_info mtd; /*分区的信息（大部分由其 master 决定）*/
3     struct mtd_info *master; /*该分区的主分区*/
4     u_int32_t offset; /*该分区的偏移地址*/
5     int index; /*分区号*/
6     struct list_head list;
7     int registered;
8 };
```

`mtd_partition` 会在 MTD 原始设备层调用 `add_mtd_partitions()` 时传递分区信息用，这个结构体的定义如代码清单 19.3 所示。

代码清单 19.3 `mtd_partition` 结构体

```
1 struct mtd_partition {
2     char *name; /* 标识字符串 */
3     u_int32_t size; /* 分区大小 */
4     u_int32_t offset; /* 主 MTD 空间内的偏移 */
5     u_int32_t mask_flags; /* 掩码标志 */
6     struct nand_ecclayout *ecclayout; /* OOB 布局 */
7     struct mtd_info **mtdp;
8 };
```

Flash 驱动中使用如下两个函数注册和注销分区：

```
int add_mtd_partitions(struct mtd_info *master, struct mtd_partition *parts, int nbparts);
int del_mtd_partitions(struct mtd_info *master);
```

`add_mtd_partitions()` 会对每一个新建分区建立一个新的 `mtd_part` 结构体，将其加入 `mtd_partitions` 中，并调用 `add_mtd_device()` 将此分区作为 MTD 设备加入 `mtd_table`。成功时返回 0，如果分配 `mtd_part` 时内存不足，则返回 `-ENOMEM`。

`del_mtd_partitions()` 的作用是针对 `mtd_partitions` 上的每一个分区，如果它的主分区是 `master`（参数 `master` 是被删除分区的主分区），则将它从 `mtd_partitions` 和 `mtd_table` 中删除并释放掉，这个函数会调用 `del_mtd_device()`。

`add_mtd_partitions()` 中新建的 `mtd_part` 需要依赖传入的 `mtd_partition` 参数对其进行初始化，如代码清单 19.4 所示。

代码清单 19.4 `add_mtd_partitions()` 函数

```
1 int add_mtd_partitions(struct mtd_info *master,
2                       const struct mtd_partition *parts,
3                       int nbparts)
4 {
5     struct mtd_part *slave;
6     u_int32_t cur_offset = 0;
7     int i;
8
9     printk(KERN_NOTICE "Creating %d MTD partitions on \"%s\":\n", nbparts, master->name);
10 }
```



```
11 for (i = 0; i < nbparts; i++) {
12     slave = add_one_partition(master, parts + i, i, cur_offset);
13     if (!slave)
14         return -ENOMEM;
15     cur_offset = slave->offset + slave->mtd.size;
16 }
17
18 return 0;
19 }
20
21 static struct mtd_part *add_one_partition(struct mtd_info *master,
22     const struct mtd_partition *part, int partno,
23     u_int32_t cur_offset)
24 {
25     struct mtd_part *slave;
26
27     /* allocate the partition structure */
28     slave = kzalloc(sizeof(*slave), GFP_KERNEL);
29     ...
30     list_add(&slave->list, &mtd_partitions);
31
32     /* 设置分区的 MTD 对象 */
33     slave->mtd.type = master->type;
34     slave->mtd.flags = master->flags & ~part->mask_flags;
35     slave->mtd.size = part->size;
36     slave->mtd.writesize = master->writesize;
37     slave->mtd.oobsize = master->oobsize;
38     slave->mtd.oobavail = master->oobavail;
39     slave->mtd.subpage_sft = master->subpage_sft;
40
41     slave->mtd.name = part->name;
42     slave->mtd.owner = master->owner;
43
44     slave->mtd.read = part_read;
45     slave->mtd.write = part_write;
46
47     ...
48
49     if (slave->offset == MTDPART_OFS_APPEND)
50         slave->offset = cur_offset;
51     if (slave->offset == MTDPART_OFS_NEXTBLK) {
52         slave->offset = cur_offset;
53         if ((cur_offset % master->erasesize) != 0) {
54             slave->offset = ((cur_offset / master->erasesize) + 1) * master->erasesize;
55             printk(KERN_NOTICE "Moving partition %d: "
56                 "0x%08x → 0x%08x\n", partno,
57                 cur_offset, slave->offset);
58         }
59     }
60     if (slave->mtd.size == MTDPART_SIZ_FULL)
61         slave->mtd.size = master->size - slave->offset;
62
63     ...
64
65     /* 注册分区 */
```

```

66  add_mtd_device(&slave_mtd);
67
68  return slave;
69 }

```

最后, 为了使系统能支持 MTD 字符设备与块设备及 MTD 分区, 在编译内核时应该包括相应的配置选项, 如下所示。

```

Memory Technology Devices (MTD)--->
<*> Memory Technology Device (MTD) support
[*] MTD partitioning support
.....
--- User Modules And Translation Layers
<*> Direct char device access to MTD devices
<*> Caching block device access to MTD devices
.....

```

19.1.3 MTD 用户空间编程

drivers/mtd/mtdchar.c 文件实现了 MTD 字符设备接口, 通过它, 用户可以直接操作 Flash 设备。通过 read()、write() 系统调用可以读写 Flash, 通过一系列 IOCTL 命令可以获取 Flash 设备信息、擦除 Flash、读写 NAND 的 OOB、获取 OOB layout 及检查 NAND 坏块等。

代码清单 19.5 所示为 MEMGETINFO、MEMERASE、MEMREADOOB、MEMWRITEOOB、MEMGETBADBLOCK IOCTL 的例子。

代码清单 19.5 /dev/mtdX IOCTL 演示范例

```

1  mtd_oob_buf oob;
2  erase_info_user erase;
3  mtd_info_user meminfo;
4
5  /*得到 MTD 设备信息*/
6  if (ioctl(fd, MEMGETINFO, &meminfo) != 0)
7      perror("ioctl(MEMGETINFO)");
8
9  /*擦除块*/
10 if (ioctl(ofd, MEMERASE, &erase) != 0) {
11     perror("ioctl(MEMERASE)");
12     goto error;
13 }
14
15 /*读 OOB */
16 if (ioctl(fd, MEMREADOOB, &oob) != 0)
17     perror("ioctl(MEMREADOOB)");
18
19 /*写 OOB */
20 if (ioctl(fd, MEMWRITEOOB, &oob) != 0) {
21     fprintf(stderr, "\n%s: %s: MTD writeoob failure: %s\n", exe_name, mtd_device,
22             strerror(errno));
23 }
24
25 /*检查坏块*/
26 if (blockstart != (ofs & (~meminfo.eraserize + 1))) {
27     blockstart = ofs & (~meminfo.eraserize + 1);
28     if ((badblock = ioctl(fd, MEMGETBADBLOCK, &blockstart)) < 0)

```



```
29     perror("ioctl(MEMGETBADBLOCK)");
30     else if (badblock) { /*是坏块*/
31         ...
32     } else /*是好块*/
33         ...
34 }
```

代码清单 19.6 所示的过程如下：它读取记录在一个映像文件中的针对 NAND 的数据信息，并把它写到 NAND Flash 中。代码中涉及大量 IOCTL 的调用及 NAND 的擦除和写入过程（含坏块检查），mtd-utils 中 `nand_write`、`flash_eraseall` 等工具也是借助类似方法实现的。

代码清单 19.6 /dev/mtdX 用户空间编程综合范例

```
1  int main(int argc, char **argv)
2  {
3      struct mtd_info_user meminfo;
4      struct mtd_oob_buf oob;
5      char oobbuf[MAX_OOB_SIZE];
6      ...
7
8      memset(oobbuf, 0xff, sizeof(oobbuf));
9
10     /* 打开/dev/mtdX */
11     if ((fd = open(mtd_device, O_RDWR)) == - 1) {
12         perror("open Flash");
13         exit(1);
14     }
15
16     /* 填充 MTD 设备容量结构体 */
17     if (ioctl(fd, MEMGETINFO, &meminfo) != 0) {
18         perror("MEMGETINFO");
19         close(fd);
20         exit(1);
21     }
22
23     oob.length = meminfo.oobsize;
24     oob.ptr = oobbuf;
25
26     /* 打开输入文件 */
27     if ((ifd = open(img, O_RDONLY)) == - 1) {
28         perror("open input file");
29         goto restoreoob;
30     }
31
32
33     imglen = lseek(ifd, 0, SEEK_END); /* 得到映像长度 */
34     lseek(ifd, 0, SEEK_SET);
35
36     pagelen = meminfo.oobblock + meminfo.oobsize; /*一页的（数据+oob）长度*/
37     ...
38
39     /* 从输入文件读数据然后写入 MTD 设备 */
40     while (imglen && (mtdoffset < meminfo.size)) {
41         /* 在擦除块之前检查是否为坏块 */
42         while (blockstart != (mtdoffset & (~meminfo.eraserize + 1))) {
```



```

43     blockstart = mtdoffset & (~meminfo.erasesize + 1);
44     offs = blockstart;
45     baderaseblock = 0;
46     if (!quiet)
47         fprintf(stdout, "Writing data to block %x\n", blockstart);
48
49     /* 检查坏块 */
50     do {
51         if ((ret = ioctl(fd, MEMGETBADBLOCK, &offs)) < 0) {
52             perror("ioctl(MEMGETBADBLOCK)");
53             goto closeall;
54         }
55         if (ret == 1) {
56             baderaseblock = 1;
57             if (!quiet)
58                 fprintf(stderr,
59                     "Bad block at %x, %u block(s) from %x will be skipped\n",
60                     (int)offs, blockalign, blockstart);
61         }
62         if (baderaseblock) {
63             mtdoffset = blockstart + meminfo.erasesize;
64         }
65         offs += meminfo.erasesize / blockalign;
66     } while (offs < blockstart + meminfo.erasesize);
67 }
68
69 readlen = meminfo.oobblock;
70
71 /* 从输入文件中读 page 数据 */
72 if ((cnt = read(ifd, writebuf, readlen)) != readlen) {
73     if (cnt == 0) /* EOF */
74         break;
75     perror("File I/O error on input file");
76     goto closeall;
77 }
78
79 /* 从输入文件读 OOB 数据 */
80 if ((cnt = read(ifd, oobreadbuf, meminfo.oobsize)) != meminfo.oobsize) {
81     perror("File I/O error on input file");
82     goto closeall;
83 }
84
85 /* 将 OOB 数据写入设备 */
86 oob.start = mtdoffset;
87 if (ioctl(fd, MEMWRITEOOB, &oob) != 0) {
88     perror("ioctl(MEMWRITEOOB)");
89     goto closeall;
90 }
91
92 /* 写 page 数据 */
93 if (pwrite(fd, writebuf, meminfo.oobblock, mtdoffset) != meminfo.oobblock) {
94     perror("pwrite");
95     goto closeall;
96 }
97 imglenn -= readlen;

```



```
98     mtdoffset += meminfo.oobblock;
99 }
100
101 closeall: ...
102 return 0;
103 }
```

19.2 NOR Flash 驱动

在 Linux 系统中, 实现了针对 CFI (公共 Flash 接口)、JEDEC (电子元件工业联合会) 等接口的通用 NOR 驱动, 这一层的驱动直接面向 `mtd_info` 的成员函数, 这使得 NOR 的芯片级驱动变得十分简单, 只需要定义具体的内存映射情况结构体 `map_info` 并使用指定接口类型调用 `do_map_probe()`。

NOR Flash 驱动的核心是定义 `map_info` 结构体, 它指定了 NOR Flash 的基址、位宽、大小等信息以及 Flash 的读写函数, 该结构体对于 NOR Flash 驱动而言非常关键, 甚至 NOR Flash 驱动的代码本质上可以被认作是根据 `map_info` 探测芯片的过程, 其定义如代码清单 19.7 所示。

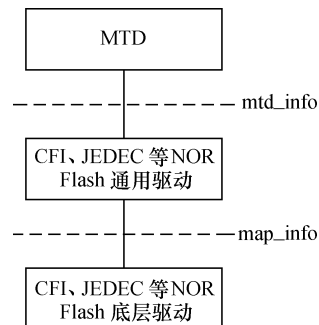


图 19.3 MTD、通用 NOR Flash 驱动与 `map_info`

代码清单 19.7 `map_info` 结构体

```
1 struct map_info {
2     char *name;
3     unsigned long size;
4     unsigned long phys;
5     #define NO_XIP (-1UL)
6
7     void __iomem *virt; /* 虚拟地址 */
8     void *cached;
9
10    int bankwidth; /* 总线宽度 */
11
12    #ifdef CONFIG_MTD_COMPLEX_MAPPINGS
13        map_word(*read)(struct map_info *, unsigned long);
14        void(*copy_from)(struct map_info *, void *, unsigned long, ssize_t);
15
16        void(*write)(struct map_info *, const map_word, unsigned long);
17        void(*copy_to)(struct map_info *, unsigned long, const void *, ssize_t);
18    #endif
19    /* 缓存的虚拟地址 */
20    void(*inval_cache)(struct map_info *, unsigned long, ssize_t);
21
22    void(*set_vpp)(struct map_info *, int);
23
24    unsigned long map_priv_1;
25    unsigned long map_priv_2;
26    void *fldrv_priv;
27    struct mtd_chip_driver *fldrv;
28 };
```

NOR Flash 驱动在 Linux 中的实现非常简单，如图 19.4 所示，主要的工作如下。

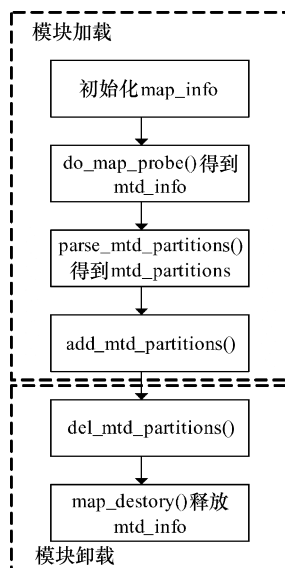


图 19.4 NOR Flash 驱动

(1) 定义 `map_info` 的实例，初始化其中的成员，根据目标板的情况为 `name`、`size`、`bankwidth` 和 `phys` 赋值。

(2) 如果 Flash 要分区，则定义 `mtd_partition` 数组，将实际电路板中 Flash 分区信息记录于其中。

(3) 以 `map_info` 和探测的接口类型（如“`cfi_probe`”、“`jedec_probe`”等）为参数调用 `do_map_probe()`，探测 Flash 得到 `mtd_info`。

`do_map_probe()` 的函数原型为：

```
struct mtd_info *do_map_probe(const char *name, struct map_info *map);
```

第一个参数为探测的接口类型，常见的调用方法如下。

```
do_map_probe("cfi_probe", &xxx_map_info);
do_map_probe("jedec_probe", &xxx_map_info);
do_map_probe("map_rom", &xxx_map_info);
```

`do_map_probe()` 会根据传入的参数 `name` 通过 `get_mtd_chip_driver()` 得到具体的 MTD 驱动，调用与接口对应的 `probe()` 函数探测设备，如代码清单 19.8 所示。

代码清单 19.8 `do_map_probe()` 函数

```
1 struct mtd_info *do_map_probe(const char *name, struct map_info *map)
2 {
3     struct mtd_chip_driver *drv;
4     struct mtd_info *ret;
5
6     drv = get_mtd_chip_driver(name); /* 通过名称获得驱动 */
7
8     if (!drv && !request_module("%s", name))
9         drv = get_mtd_chip_driver(name);
10
11     if (!drv)
12         return NULL;
13 }
```



```
14 ret = drv->probe(map); /* 调用驱动的探测函数 */
15
16 module_put(drv->module);
17 if (ret)
18     return ret;
19
20 return NULL;
21 }
```

利用 `map_info` 中的配置, `do_map_probe()` 可以自动识别支持 CFI 或 JEDEC 接口的 Flash 芯片, MTD 以后会自动采用适当的命令参数对 Flash 进行读写或擦除。

(4) 在模块初始化时以 `mtd_info` 为参数调用 `add_mtd_device()` 或以 `mtd_info`、`mtd_partition` 数组及分区数为参数调用 `add_mtd_partitions()` 注册设备或分区。当然, 在这之前可以调用 `parse_mtd_partitions()` 查看 Flash 上是否已有分区信息, 并将查看出的分区信息通过 `add_mtd_partitions()` 注册。

(5) 在模块卸载时调用第 4 行函数的“反函数”删除设备或分区。

代码清单 19.9 所示为一个最简单的 NOR Flash 驱动模板。

代码清单 19.9 NOR Flash 设备驱动模板

```
1 #define WINDOW_SIZE ...
2 #define WINDOW_ADDR ...
3 static struct map_info xxx_map = { /*map_info */
4     .name = "xxx Flash",
5     .size = WINDOW_SIZE, /*大小*/
6     .bankwidth = 1, /*总线宽度*/
7     .phys = WINDOW_ADDR /*物理地址*/
8 };
9
10 static struct mtd_partition xxx_partitions[] = { /* mtd_partition */
11 {
12     .name = "Drive A",
13     .offset = 0, /*分区的偏移地址*/
14     .size = 0x0e0000 /*分区大小*/
15 },
16 ...
17 };
18
19 #define NUM_PARTITIONS ARRAY_SIZE(xxx_partitions)
20
21 static struct mtd_info *mymtd;
22
23 static int __init init_xxx_map(void)
24 {
25     int rc = 0;
26
27     xxx_map.virt=ioremap_nocache(xxx_map.phys, xxx_map.size); /*物理→虚拟地址*/
28
29     if (!xxx_map.virt) {
30         printk(KERN_ERR "Failed to ioremap_nocache\n");
31         rc = -EIO;
32         goto err2;
33     }
34
35     simple_map_init(&xxx_map);
36 }
```

```

37 mymtd = do_map_probe("jedec_probe", &xxx_map); /*探测 nor Flash */
38 if (!mymtd) {
39     rc = -ENXIO;
40     goto err1;
41 }
42
43 mymtd->owner = THIS_MODULE;
44 add_mtd_partitions(mymtd, xxx_partitions, NUM_PARTITIONS); /*添加分区信息*/
45
46 return 0;
47
48 err1:
49 map_destroy(mymtd);
50 iounmap(xxx_map.virt);
51 err2:
52 return rc;
53 }
54
55 static void __exit cleanup_xxx_map(void)
56 {
57     if (mymtd) {
58         del_mtd_partitions(mymtd); /*删除分区*/
59         map_destroy(mymtd);
60     }
61
62     if (xxx_map.virt) {
63         iounmap(xxx_map.virt);
64         xxx_map.virt = NULL;
65     }
66 }

```

19.3 NAND Flash 驱动

和 NOR Flash 非常类似，如图 19.5 所示，Linux 内核在 MTD 的下层实现了通用的 NAND 驱动（主要通过 `drivers/mtd/nand/nand_base.c` 文件实现），因此芯片级的 NAND 驱动不再需要实现 `mtd_info` 中的 `read()`、`write()`、`read_oob()`、`write_oob()` 等成员函数，而主体转移到了 `nand_chip` 数据结构。

MTD 使用 `nand_chip` 数据结构表示一个 NAND Flash 芯片，这个结构体中包含了关于 NAND Flash 的地址信息、读写方法、ECC 模式、硬件控制等一系列底层机制，其定义如代码清单 19.10 所示。

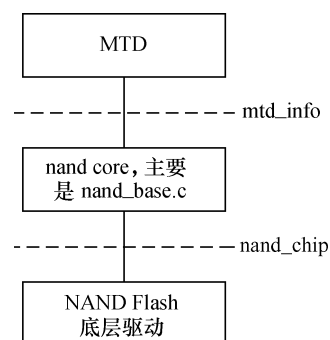


图 19.5 NAND Flash 驱动

代码清单 19.10 `nand_chip` 结构体

```

1 struct nand_chip {
2     void __iomem *IO_ADDR_R; /* 读 8 位 I/O 线的地址，由板决定 */
3     void __iomem *IO_ADDR_W; /* 写 8 位 I/O 线的地址，由板决定 */
4

```



```
5  uint8_t      (*read_byte)(struct mtd_info *mtd);
6  ul6         (*read_word)(struct mtd_info *mtd);
7  void        (*write_buf)(struct mtd_info *mtd, const uint8_t *buf, int len);
8  void        (*read_buf)(struct mtd_info *mtd, uint8_t *buf, int len);
9  int         (*verify_buf)(struct mtd_info *mtd, const uint8_t *buf, int len);
10 void        (*select_chip)(struct mtd_info *mtd, int chip); /* 片选芯片 */
11 int         (*block_bad)(struct mtd_info *mtd, loff_t ofs, int getchip); /* 是否坏块 */
12 int         (*block_markbad)(struct mtd_info *mtd, loff_t ofs); /* 标记坏块 */
13 void        (*cmd_ctrl)(struct mtd_info *mtd, int dat,
14                      unsigned int ctrl); /* 控制 ALE/CLE/nCE, 也用于写命令和地址 */
15 int         (*dev_ready)(struct mtd_info *mtd); /* 设备就绪 */
16 void        (*cmdfunc)(struct mtd_info *mtd, unsigned command, int column, int page_addr);
17 int         (*waitfunc)(struct mtd_info *mtd, struct nand_chip *this);
18 void        (*erase_cmd)(struct mtd_info *mtd, int page);
19 int         (*scan_bbt)(struct mtd_info *mtd);
20 int         (*errstat)(struct mtd_info *mtd, struct nand_chip *this, int state, int status, int page);
21 int         (*write_page)(struct mtd_info *mtd, struct nand_chip *chip,
22                      const uint8_t *buf, int page, int cached, int raw);
23
24 int         chip_delay;
25 unsigned int options;
26
27 int         page_shift;
28 int         phys_erase_shift;
29 int         bbt_erase_shift;
30 int         chip_shift;
31 int         numchips;
32 unsigned long chipsize;
33 int         pagemask;
34 int         pagebuf;
35 int         subpagesize;
36 uint8_t      cellinfo;
37 int         badblockpos;
38
39 nand_state_t state;
40
41 uint8_t      *oob_poi;
42 struct nand_hw_control *controller;
43 struct nand_ecclayout *ecclayout;
44
45 struct nand_ecc_ctrl ecc;
46 struct nand_buffers *buffers;
47 struct nand_hw_control hwcontrol;
48
49 struct mtd_oob_ops ops;
50
51 uint8_t      *bbt;
52 struct nand_bbt_descr *bbt_td;
53 struct nand_bbt_descr *bbt_md;
54
55 struct nand_bbt_descr *badblock_pattern;
56
57 void        *priv;
58 };
```

与 NOR Flash 类似, 由于有了 MTD 层, 完成一个 NAND Flash 驱动在 Linux 中的工作量也很小, 如图 19.6 所示, 主要的工作如下。

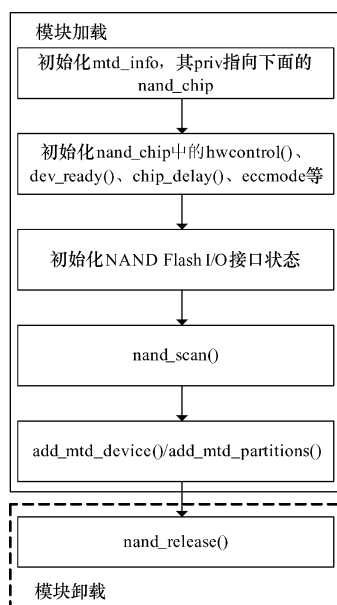


图 19.6 NAND Flash 驱动

(1) 如果 Flash 要分区, 则定义 `mtd_partition` 数组, 将实际电路板中 Flash 分区信息记录于其中。

(2) 在模块加载时分配和 `nand_chip` 的内存, 根据目标板 NAND 控制器的情况初始化 `nand_chip` 中的 `cmd_ctrl()`、`dev_ready()`、`read_byte()`、`read_buf()`、`write_buf()`、`select_chip()`、`block_bad()`、`block_markbad()` 等成员函数 (如果不赋值会使用 `nand_base.c` 中的默认函数, 这里典型利用了面向对象的继承和重载的思想), 注意将 `mtd_info` 的 `priv` 置为 `nand_chip`。

(3) 以 `mtd_info` 为参数调用 `nand_scan()` 函数探测 NAND Flash 的存在。`nand_scan()` 函数的原型为:

```
int nand_scan (struct mtd_info *mtd, int maxchips);
```

`nand_scan()` 函数会读取 NAND 芯片 ID, 并根据 `mtd→priv` 即 `nand_chip` 中的成员初始化 `mtd_info`。

(4) 如果要分区, 则以 `mtd_info` 和 `mtd_partition` 为参数调用 `add_mtd_partitions()`, 添加分区信息。代码清单 19.11 所示为一个简单的 NAND Flash 设备驱动模板。

代码清单 19.11 NAND Flash 设备驱动模板

```

1 #define CHIP_PHYSICAL_ADDRESS ...
2 #define NUM_PARTITIONS 2
3 static struct mtd_partition partition_info[] = {
4     {
5         .name = "Flash partition 1", .offset = 0, .size = 8 * 1024 * 1024
6     },
7     {
8         .name = "Flash partition 2", .offset = MTDPART_OFS_NEXT, .size =
9         MTDPART_SIZ_FULL
10    },
11 };
12 int __init board_init(void)
13 {
14     struct nand_chip *this;
15     int err = 0;
16     /* 为 MTD 设备结构体和 nand_chip 分配内存 */
  
```



```
17 board_mtd = kmalloc(sizeof(struct mtd_info) + sizeof(struct nand_chip),
18     GFP_KERNEL);
19 if (!board_mtd) {
20     printk("Unable to allocate NAND MTD device structure.\n");
21     err = - ENOMEM;
22     goto out;
23 }
24 /* 初始化结构体 */
25 memset((char*)board_mtd, 0, sizeof(struct mtd_info) + sizeof(struct nand_chip));
26 /* 映射物理地址 */
27 baseaddr = (unsigned long)ioremap(CHIP_PHYSICAL_ADDRESS, 1024);
28 if (!baseaddr) {
29     printk("Ioremap to access NAND chip failed\n");
30     err = - EIO;
31     goto out_mtd;
32 }
33 /* 获得私有数据 (nand_chip) 指针 */
34 this = (struct nand_chip*)&board_mtd[1];
35 /* 将 nand_chip 赋予 mtd_info 私有指针 */
36 board_mtd->priv = this;
37 /* 设置 NAND Flash 的 I/O 基地址 */
38 this->IO_ADDR_R = baseaddr;
39 this->IO_ADDR_W = baseaddr;
40 /* 硬件控制函数 */
41 this->cmd_ctrl = board_hwcontrol;
42 /* 初始化设备 ready 函数 */
43 this->dev_ready = board_dev_ready;
44 /* 扫描以确定设备的存在 */
45 if (nand_scan(board_mtd, 1)) {
46     err = - ENXIO;
47     goto out_ior;
48 }
49 /* 添加分区 */
50 add_mtd_partitions(board_mtd, partition_info, NUM_PARTITIONS);
51 goto out;
52 out_ior: iounmap((void*)baseaddr);
53 out_mtd: kfree(board_mtd);
54 out: return err;
55 }
56
57 static void __exit board_cleanup(void)
58 {
59     /* 释放资源, 注销设备 */
60     nand_release(board_mtd);
61     /* unmap 物理地址 */
62     iounmap((void*)baseaddr);
63     /* 释放 MTD 设备结构体 */
64     kfree(board_mtd);
65 }
66
67 /* 硬件控制 */
68 static void board_hwcontrol(struct mtd_info *mtd, int dat, unsigned int ctrl)
69 {
70     ...
71     if (ctrl & NAND_CTRL_CHANGE) {
72         if (ctrl & NAND_NCE) {
73             ...
```



```

74 }
75
76 /* 返回设备 ready 状态 */
77 static int board_dev_ready(struct mtd_info *mtd)
78 {
79     return xxx_read_ready_bit();
80 }

```

最后要强调的是, 在 NAND 芯片级驱动中, 如果在 `nand_chip` 中没有赋值, 将使用如代码清单 19.12 所示的默认分布。因此, 若不使用默认分布, 在 NAND 驱动中, 应该根据实际 NAND 控制器和 NAND 芯片的情况给 `nand_chip` 的 `nand_ecc_ctrl` 结构体类型成员 `ecc` 赋值, 定义 OOB 的分布和模式。

代码清单 19.12 NAND 驱动默认的 OOB 分布

```

1  /* 页大小为 256、512、2K、4K 字节情况下默认的 ECC 布局 */
2
3  static struct nand_ecclayout nand_oob_8 = {
4      .eccbytes = 3,
5      .eccpos = {0, 1, 2},
6      .oobfree = {
7          {.offset = 3,
8           .length = 2},
9          {.offset = 6,
10         .length = 2}}
11 };
12
13 static struct nand_ecclayout nand_oob_16 = {
14     .eccbytes = 6,
15     .eccpos = {0, 1, 2, 3, 6, 7},
16     .oobfree = {
17         {.offset = 8,
18          .length = 8}}
19 };
20
21 static struct nand_ecclayout nand_oob_64 = {
22     .eccbytes = 24,
23     .eccpos = {
24         40, 41, 42, 43, 44, 45, 46, 47,
25         48, 49, 50, 51, 52, 53, 54, 55,
26         56, 57, 58, 59, 60, 61, 62, 63},
27     .oobfree = {
28         {.offset = 2,
29          .length = 38}}
30 };
31
32 static struct nand_ecclayout nand_oob_128 = {
33     .eccbytes = 104,
34     .eccpos = {
35         24, 25, 26, 27, 28, 29, 30, 31,
36         32, 33, 34, 35, 36, 37, 38, 39,
37         40, 41, 42, 43, 44, 45, 46, 47,
38         48, 49, 50, 51, 52, 53, 54, 55,
39         56, 57, 58, 59, 60, 61, 62, 63,
40         64, 65, 66, 67, 68, 69, 70, 71,
41         72, 73, 74, 75, 76, 77, 78, 79,
42         80, 81, 82, 83, 84, 85, 86, 87,
43         88, 89, 90, 91, 92, 93, 94, 95,
44         96, 97, 98, 99, 100, 101, 102, 103,

```



```
45         104, 105, 106, 107, 108, 109, 110, 111,  
46         112, 113, 114, 115, 116, 117, 118, 119,  
47         120, 121, 122, 123, 124, 125, 126, 127},  
48     .oobfree = {  
49         {.offset = 2,  
50          .length = 22}}  
51 };
```

上述代码中的 `eccpos` 表明 ECC 校验码在 OOB 区域的存放位置, `eccbytes` 是校验码的长度, `oobfree` 则是除了校验码之外可用的 OOB 字节。

`nand_ecc_ctrl` 结构体的 `mode` 字段定义 ECC 的放置模式, 包括 `MTD_NANDECC_OFF` (不使用 ECC)、`NAND_ECC_SOFT` (使用软件 ECC)、`NAND_ECC_HW` (使用硬件 ECC) 等。如果 NAND 控制器支持通过硬件进行 ECC 校验, 则最好使用 `NAND_ECC_HW`。

内核中包含了一个 NAND 模拟器 `nandsim`, 使用一片内存区域模拟 NAND, 在没有电路板的情况下, 可以使用 `nandsim` 模拟 NAND 芯片。YAFFS 和 YAFFS2 中分别包含了 `nandemul` 和 `nandemul2k` (用于模拟页大小为 2KB 的 NAND), 也可以模拟 NAND。

19.4 NOR Flash 驱动实例 :S3C6410 外围的 NOR Flash 驱动

针对 S3C2410、S3C6410 等平台而言, 外接 NOR Flash 的情况下, 由于该 NOR Flash 直接映射在 CPU 的内存空间上, 因此可以直接使用通用的 `drivers/mtd/maps/physmap.c` 驱动, 在内核配置的时候应该使能 `MTD_PHYSMAP`。为了使用 NOR Flash, 我们只需要在 BSP 的板文件中添加相应的信息, 如 NOR Flash 所在的物理地址和大小、分区信息、总线宽度等, 这些信息以 `platform` 资源和数据的形式呈现, 如代码清单 19.13。

代码清单 19.13 S3C6410 外围 NOR Flash 的 `platform` 数据

```
1 static struct resource ldd6410_nor_resource = {  
2     .start = LDD6410_NOR_BASE,  
3     .end   = LDD6410_NOR_BASE + 0x200000 - 1,  
4     .flags = IORESOURCE_MEM,  
5 };  
6  
7 static struct mtd_partition ldd6410_mtd_partitions[] = {  
8     {  
9         .name      = "System",  
10        .size       = 0x40000,  
11        .offset     = 0,  
12        .mask_flags = MTD_WRITEABLE, /* force read-only */  
13    }, {  
14        .name      = "Data",  
15        .size       = 0x1C0000,  
16        .offset     = MTDPART_OFST_APPEND,  
17    },  
18 };  
19  
20 static struct physmap_flash_data ldd6410_flash_data = {  
21     .width      = 2,  
22     .parts       = ldd6410_mtd_partitions,
```

```

23     .nr_parts      = ARRAY_SIZE(ldd6410_mtd_partitions),
24 };
25
26 static struct platform_device ldd6410_device_nor = {
27     .name           = "physmap-flash",
28     .id             = 0,
29     .dev = {
30         .platform_data = &ldd6410_flash_data,
31     },
32     .num_resources  = 1,
33     .resource       = &ldd6410_nor_resource,
34 };

```

上述代码第 27 行指定 platform 设备的名称为“physmap-flash”，这和对应 platform 驱动 drivers/mtd/maps/physmap.c 中定义的名称是一致的。

19.5 NAND Flash 驱动实例：S3C6410 外围的 NAND Flash 驱动

19.5.1 S3C6410 NAND 控制器硬件描述

S3C6410 处理器集成了一个 NAND 控制器，它支持页大小为 512 字节和 2048 字节的 SLC 或 MLC NAND Flash。对 SLC 工艺 Flash，支持 1-bit 的硬件 ECC，对 MLC 工艺 Flash，支持 4-bit 或 8-bit 的硬件 ECC。

LDD6410 开发板连接了一块 K9F2G08 的 NAND Flash，其原理如图 19.7 所示，使用的驱动是 drivers/mtd/nand/s3c_nand.c，它同时支持 S3C64XX、S5P64XX、S5PC1XX 处理器，对应的内核配置选项为 MTD_NAND_S3C，如果要使用硬件 ECC 功能，还需要使能 MTD_NAND_S3C_HWECCE。

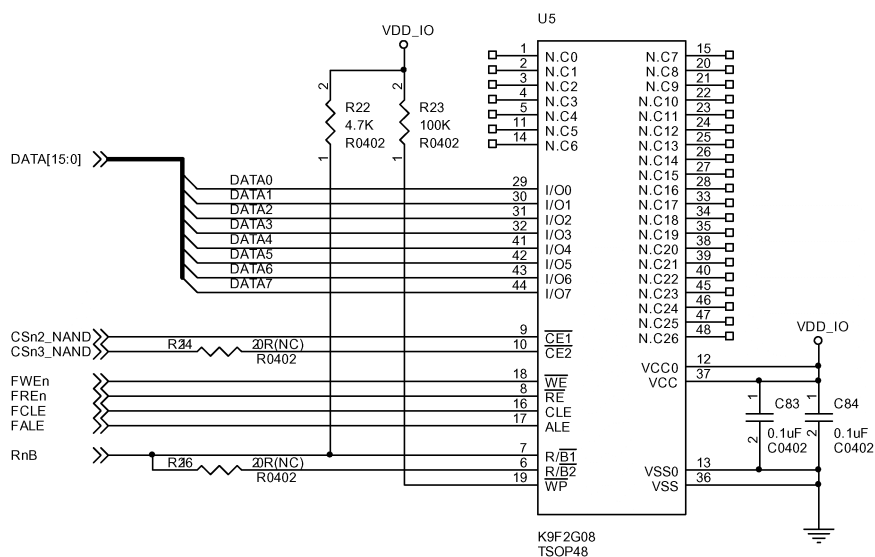


图 19.7 LDD6410 开发板上的 NAND 连接原理



19.5.2 S3C6410 nand_chip 初始化与 NAND 探测

S3C6410 的 NAND 驱动以 platform 驱动的形式存在, 在执行 probe() 时, 初始化 nand_chip 实例并运行 nand_scan() 扫描 NAND 设备, 最后调用 add_mtd_partitions() 添加板文件 platform 中定义的分表。nand_chip 是 NAND Flash 驱动的核心数据结构, 这个结构体中的成员直接对应着 NAND Flash 的底层操作, 针对具体的 NAND 控制器情况, 本驱动中初始化了 IO_ADDR_R、IO_ADDR_W、cmd_ctrl()、dev_ready()、scan_bbt() 以及 ECC 相关的信息。代码清单 19.14 所示为 S3C6410 外围 NAND Flash 驱动的 nand_chip 初始化与注册过程。

代码清单 19.14 S3C6410 nand_chip 初始化与注册

```
1 static int s3c_nand_probe(struct platform_device *pdev, enum s3c_cpu_type cpu_type)
2 {
3     ...
4     for (i = 0; i < plat_info->chip_nr; i++) {
5         nand->IO_ADDR_R = (char *) (s3c_nand.regs + S3C_NFDATA);
6         nand->IO_ADDR_W = (char *) (s3c_nand.regs + S3C_NFDATA);
7         nand->cmd_ctrl = s3c_nand_hwcontrol;
8         nand->dev_ready = s3c_nand_device_ready;
9         nand->scan_bbt = s3c_nand_scan_bbt;
10        nand->options = 0;
11
12        #if defined(CONFIG_MTD_NAND_S3C_CACHEDPRG)
13            nand->options |= NAND_CACHEPRG;
14        #endif
15
16        #if defined(CONFIG_MTD_NAND_S3C_HWECCE)
17            nand->ecc.mode = NAND_ECC_HW;
18            nand->ecc.hwctl = s3c_nand_enable_hwecc;
19            nand->ecc.calculate = s3c_nand_calculate_ecc;
20            nand->ecc.correct = s3c_nand_correct_data;
21
22            s3c_nand_hwcontrol(0, NAND_CMD_READID, NAND_NCE | NAND_CLE | NAND_CTRL_CHANGE);
23            s3c_nand_hwcontrol(0, 0x00, NAND_CTRL_CHANGE | NAND_NCE | NAND_ALE);
24            s3c_nand_hwcontrol(0, 0x00, NAND_NCE | NAND_ALE);
25            s3c_nand_hwcontrol(0, NAND_CMD_NONE, NAND_NCE | NAND_CTRL_CHANGE);
26            s3c_nand_device_ready(0);
27
28            tmp = readb(nand->IO_ADDR_R); /* 制造商 ID */
29            tmp = readb(nand->IO_ADDR_R); /* 设备 ID */
30            devID = tmp;
31
32            for (j = 0; nand_flash_ids[j].name != NULL; j++) {
33                if (tmp == nand_flash_ids[j].id) {
34                    type = &nand_flash_ids[j];
35                    break;
36                }
37            }
38
39            ...
40            nand->cellinfo = readb(nand->IO_ADDR_R); /* 第 3 个字节 */
41            tmp = readb(nand->IO_ADDR_R); /* 第 4 个字节 */
42
43            if (!type->pagesize) {
```

```

44         if ((nand->cellinfo >> 2) & 0x3) == 0) {
45             nand_type = S3C_NAND_TYPE_SLC;
46             nand->ecc.size = 512;
47             nand->ecc.bytes = 4;
48             if (devID == 0xd5) {
49                 /* Page size is 4Kbytes */
50                 nand->ecc.read_page = s3c_nand_read_page_8bit;
51             ...
52             } else {
53                 if ((1024 << (tmp & 3)) == 4096) /* For 4KB Page 8_bit ECC */
54                 {
55                     /* Page size is 4Kbytes */
56                     nand->ecc.read_page = s3c_nand_read_page_8bit;
57                 ...
58                 } else {
59                     ...
60                 }
61             }
62         } else {
63             nand_type = S3C_NAND_TYPE_MLC;
64             nand->options |= NAND_NO_SUBPAGE_WRITE; /* NOP = 1 if MLC */
65             if (devID == 0xd5) {
66                 /* Page size is 4Kbytes */
67                 nand->ecc.read_page = s3c_nand_read_page_8bit;
68                 nand->ecc.write_page = s3c_nand_write_page_8bit;
69                 ...
70             } else {
71                 if ((1024 << (tmp & 3)) == 4096) {
72                     /* Page size is 4Kbytes */
73                     nand->ecc.read_page = s3c_nand_read_page_8bit;
74                     nand->ecc.write_page = s3c_nand_write_page_8bit;
75                     nand->ecc.read_oob = s3c_nand_read_oob_8bit;
76                     ...
77                 } else {
78                     nand->ecc.read_page = s3c_nand_read_page_4bit;
79                     ...
80                 }
81             }
82         }
83     ...
84
85     printk("S3C NAND Driver is using hardware ECC.\n");
86 #else
87     nand->ecc.mode = NAND_ECC_SOFT;
88     printk("S3C NAND Driver is using software ECC.\n");
89 #endif
90     if (nand_scan(s3c_mtd, 1)) {
91         ret = -ENXIO;
92         goto exit_error;
93     }
94
95     /* 注册分区信息 */
96     add_mtd_partitions(s3c_mtd, partition_info, plat_info->mtd_part_nr);
97 }
98

```



```
99 pr_debug("initialized ok\n");
100 return 0;
101 ...
102 }
```

drivers/mtd/nand/s3c_nand.c 是一个 platform 驱动, 我们在 LDD6410 的 BSP 中只需要添加相关的针对 NAND 的 platform 设备和分区信息即可。

19.6 Flash 文件系统的建立

19.6.1 Flash 转换层

由于无法重复地在 Flash 的同一块存储位置做写入操作 (必须事先擦除该块后才能再写入), 因此一般在硬盘上使用的文件系统, 如 VFAT、NTFS、EXT2、EXT3 等将无法直接用在 Flash 上, 为了沿用这些文件系统, 则必须透过一层转换层 (Translation Layer) 来将逻辑块地址 (Logical Block Address) 对应到 Flash 存储器的物理位置, 使系统能把 Flash 当作普通的硬盘一样处理, 我们称这层为 FTL (Flash Translation Layer)。FTL 应用于 NOR Flash, 而 NFTL 则应用于 NAND Flash, 如图 19.8 所示。

一个闪存转换层的最简单的实现就是将模拟的块设备一对一地映射到闪存上。举例来说, 当上层的文件系统要写一个块设备的扇区时, 闪存转换层要做下面的操作来完成这个写请求。

- (1) 将这个扇区所在擦除块的数据读到内存中, 放在缓存中。
- (2) 将缓存中与这个扇区对应的内容用新的内容替换。
- (3) 对该擦除块执行擦除操作。
- (4) 将缓冲中的数据写回该擦除块。

这种实现方式的缺点如下。

- 效率低, 对一个扇区的更新要重写整个擦除块上的数据, 造成数据带宽很大的浪费。可行的办法是只有当文件系统的写请求超过了一个擦除块的边界的时候, 才去执行对闪存的擦除、写回操作 (这种更新方式也叫 out-of-place)。
- 没有提供磨损平衡, 那些被频繁更新的数据所在擦除块将首先变成坏块。
- 非常不安全, 很容易引起数据的丢失。如果在上面的第 (3) 步和第 (4) 步之间发生了突然掉电, 那么整个擦除块中的数据就全部丢失了。

为了解决上面这种实现方式的问题, 闪存转换层不能只是简单地实现块设备与闪存的一一映射, 它还需要将模拟块设备的扇区存储在闪存的不同位置, 并且维持扇区到闪存的映射关系。而

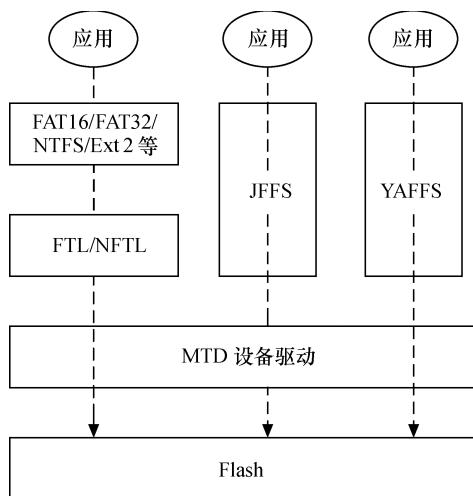


图 19.8 FTL 和 NFTL

为了进行垃圾回收 (Garbage Collection)，闪存转换层必须能理解上层文件系统的语义。这样实现导致的最大问题就是效率不高，具体来说，闪存转换层为了能理解上层文件系统的语义，必须对文件系统的每个写请求进行解析，因此导致写操作的性能下降。另外，从软件的架构上来讲，要求文件系统下面的一层去理解文件系统的语义，也不太合理。因此，在 Flash 上，应尽可能地避免使用传统的依赖闪存转换层的文件系统，最好应采用专门的针对 Flash 的文件系统。

19.6.2 CramFS

在嵌入式 Linux 环境中，许多人会采用 RAMDISK 来储存文件系统的内容，RAMDISK 的含义是在启动时，把一部份内存虚拟成磁盘，并且把之前准备好的文件系统映像文件解压缩到该 RAMDISK 环境中。假设压缩后的文件系统映像为 8MB，存放于 Flash，解压缩后为 16MB，如果采用 RAMDISK，将需要 8MB 的 Flash 和 16MB 的 RAM 空间，而采用 CramFS 后，就不再需要消耗 16MB 的 RAM 空间。

CramFS 是 Linus Torvalds 参与开发的文件系统，在 linux/fs/cramfs 中可以找到 CramFS 的源代码。CramFS 是一种压缩的只读文件系统，当浏览 Flash 中的目录或读取文件时，CramFS 文件系统会动态地计算出压缩后的数据所储存的位置，并实时地解压缩到内存中，对于用户来说，使用 CramFS 与 RAMDISK 感觉不出使用上的差异性。

CramFS 工具的下载地址为 <http://sourceforge.net/projects/cramfs/>，通过如下命令可以创建 CramFS 文件系统映像：

```
mkcramfs my_cramfs/ cramfs.img (my_cramfs 是我们要创建映像的目录)
```

如下命令将生成的 cramfs.img 映像复制到 Flash 的第一个分区并 mount 到 /mnt/nor 目录：

```
cp cramfs.img /dev/mtd1
mount -cramfs /dev/mtdblock1 /mnt/nor
```

很多时候，工程中需要基于已有的文件系统映像添加、删除一些文件后建立新的文件系统映像，这时候并不需要完全重新操作，可用如下的方法。

(1) 将映像以 loop 方式挂载到某目录。

```
mkdir tmpdir
mount rootfs.cramfs tmpdir -o loop
cd tmpdir
```

(2) 压缩被挂载的文件系统。

```
tar -cvf ../rootfs.tar ./      将 tmpdir 中的内容打包放在其父目录下
umount tmpdir
```

(3) 解压缩文件系统到新目录。

```
mkdir rootfs
tar -xvf rootfs.tar -C rootfs
```

(4) 修改新目录 (这里是 rootfs) 中的内容，以符合新的需要。

(5) 重新创建映像文件。

```
mkcramfs rootfs rootfs.cramfs
```

19.6.3 JFFS/JFFS2

JFFS 是由瑞典 Axis Communications AB 公司开发的，于 1999 年末基于 GNU GPL 发布的文件系统。最初的发布版本基于 Linux 2.0，后来 Red Hat 将它移植到 Linux 2.2，在使用的过程中，



JFFS 设计中的局限被不断地暴露出来。于是在 2001 年初, Red Hat 决定实现一个新的 JFFS2 (<http://www.infradead.org>)。

JFFS2 是一个日志结构 (log-structured) 的文件系统, 它在闪存上顺序地存储包含数据和原数据 (meta-data) 的节点。JFFS2 的日志结构存储方式使得它能对闪存进行 out-of-place 更新, 而不是磁盘所采用的 in-place 更新方式。它提供的垃圾回收机制, 使得我们不需要马上对擦写越界的块进行擦写, 而只需要对其设置一个标志, 标明为“脏”块。当可用的块数不足时, 垃圾回收机制才开始回收这些节点。同时, 由于 JFFS2 基于日志结构, 在意外掉电后仍然可以保持数据的完整性, 而不会丢失数据。因此, JFFS2 成为了目前 Flash 上应用最广泛的文件系统。

然而, JFFS2 挂载时需要扫描整块 Flash 以确定节点的合法性以及建立必要的数据结构, 这使得 JFFS2 挂载时间比较长。又由于 JFFS2 将节点信息保存在内存中, 使得它所占用的内存量和节点数目成正比。再者, 由于 JFFS2 通过随机方式来实现磨损平衡, 它不能保证磨损平衡的确定性。因此, 人们提出了 JFFS3, 它就是为解决 JFFS2 的这些缺陷而设计的。

和 CramFS 一样, 也存在一个制作 JFFS2 文件系统的工具 mkfs.jffs2 (包含在 mtd-utils 中), 执行如下命令即可生成所要的映像:

```
./mkfs.jffs2 -d my-jffs2/ -o jffs2.img (my-jffs2 是我们要制作映像的目录)
```

使用 mkfs.jffs2 制作映像的时候, 要注意指定正确的擦除块大小和页面大小, 对于 NAND Flash, 应使用“-n”去掉生成映像中的 clean marker。

接下来将 jffs2.img 复制到 Flash 第 1 个分区 (复制到 MTD 字符设备), 如下所示:

```
cp jffs2.img /dev/mtd1
```

之后, 就可以将对应的块设备 mount 到 Linux 的目录了, 如下所示:

```
mount -t jffs2 /dev/mtdblock1 /mnt/nor
```

对于 NAND Flash, 应使用 mtd-utils 中的 nandwrite 工具进行 jffs2 映像向 NAND Flash 的烧录, 在烧录前可以使用 flash_eraseall 擦除 Flash, 并在 OOB 区域加上 JFFS2 需要的 clean marker。

LDD6410 开发板的文件系统中已包含 mtd-utils 系列工具。mtd-utils 的当前最新版本为 1.3.1, 其下载地址为: <ftp://ftp.infradead.org/pub/mtd-utils/mtd-utils-1.3.1.tar.bz2>。

19.6.4 YAFFS/YAFFS2

YAFFS (Yet Another Flash File System, <http://www.yaffs.net>) 文件系统是专门针对 NAND 闪存设计的嵌入式文件系统, 目前有 YAFFS 和 YAFFS2 两个版本, 两个版本的主要区别之一在于 YAFFS2 能够更好地支持大容量的 NAND Flash 芯片, 而前者只针对页大小为 512 字节的 NAND。

YAFFS 文件系统有些类似于 JFFS/JFFS2 文件系统, 与之不同的是 JFFS1/2 文件系统最初是针对 NOR Flash 的应用场合设计的, 而 NOR Flash 和 NAND Flash 本质上有较大的区别, 所以尽管 JFFS1/2 文件系统也能应用于 NAND Flash, 但由于它在内存占用和启动时间方面针对 NOR 的特性做了一些取舍, 所以对 NAND 来说通常并不是最优的方案。NAND 上的每一页数据都有额外的空间用来存储附加信息, YAFFS 正好利用了该空间中一部分来存储文件系统相关的内容。

YAFFS 和 JFFS 都提供了写均衡、垃圾收集等底层操作, 它们的不同之处如下。

- JFFS 是一种日志文件系统, 通过日志机制保证文件系统的稳定性。YAFFS 仅仅借鉴了

日志系统的思想，不提供日志机能，所以稳定性不如 JFFS，但是资源占用少。

- JFFS 中使用多级链表管理需要回收的脏块，并且使用系统生成伪随机变量决定要回收的块，通过这种方法能提供较好的写均衡，在 YAFFS 中是从头到尾对块搜索，所以在垃圾收集上 JFFS 的速度慢，但是能延长 NAND 的寿命。
- JFFS 支持文件压缩，适合存储容量较小的系统；YAFFS 不支持压缩，更适合存储容量较大的系统。
- YAFFS 还带有 NAND 芯片驱动，并为嵌入式系统提供了直接访问文件系统的 API，用户可以不使用 Linux 中的 MTD 和 VFS，直接对文件进行操作（如图 19.8）。尽管如此，NAND Flash 大多还是采用 MTD+YAFFS 的模式。

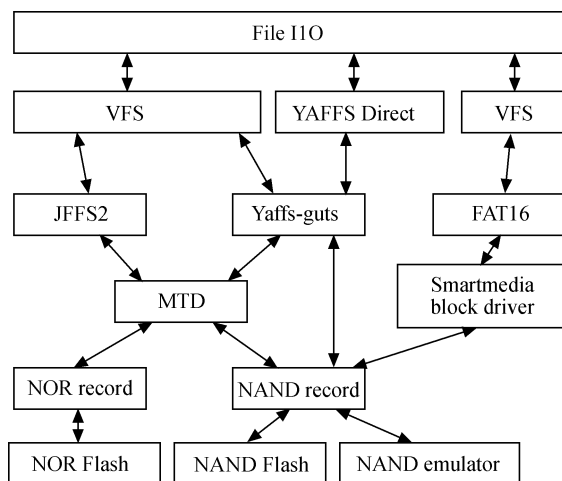


图 19.9 JFFS2 和 YAFFS

- YAFFS 使用 OOB 组织文件结构信息，而 JFFS 直接将节点信息保存在 NAND 数据区域里面，因此 YAFFS 在 mount 时只需读取 OOB，其 mount 时间远小于 JFFS。

在 Linux 2.6 内核下，YAFFS 文件系统的移植非常简单，主要包括以下工作。

(1) 复制 YAFFS 源代码到 Linux 源码树。

只需要在内核中建立 YAFFS 目录 fs/yaffs，并把下载的 YAFFS 代码复制到该目录下面（YAFFS 代码包括 yaffs_ecc.c、yaffs_fileem.c、yaffs_fs.c、yaffs_guts.c、yaffs_mtdif.c、yaffs_ramem.c），下载的 YAFFS 源代码已包含了 Kconfig 和 Makefile，因此只需要修改 fs 目录下的 Kconfig 和 Makefile 并引用 fs/yaffs 中的对应文件即可，方法是：在 fs/Kconfig 中增加 source "fs/yaffs/Kconfig"；在 fs/Makefile 中增加 obj-\$(CONFIG_YAFFS_FS) += yaffs/。

(2) 配置内核编译选项。

在采用 make menuconfig 等方式配置内核编译选项时，除了应该选中 MTD 系统及目标板上 Flash 的驱动以外，也必须选中对 YAFFS 的支持，如下所示：

```
File systems --->
Miscellaneous filesystems --->
<*> Yet Another Flash Filing System(YAFFS) file system support
[*] NAND mtd support
[*] Use ECC functions of the generic MTD-NAND driver
```



```
[*] Use Linux file caching layer
[*] Turn off debug chunk erase check
[*] Cache short names in RAM
```

(3) 挂载 YAFFS。

YAFFS 源代码包的 `utils` 目录下包含了 `mkyaffs` 工具, 可以用它格式化 Flash, 例如运行 `mkyaffs /dev/mtd3` 将用 YAFFS 文件系统格式化 NAND 的第 3 个分区, 之后我们可以运行如下命令挂载 YAFFS:

```
mount -t yaffs /dev/mtdblock3 /mnt/nand
```

此后, 对 `/mnt/nand` 的操作就是对 `/dev/mtdblock3` 的操作。

如果运行如下命令将根文件系统复制到 `/mnt/Flash0`:

```
mount -t yaffs /dev/mtdblock3 /mnt/nand
cp (our_rootfs) /mnt/Flash0
umount /mnt/nand
```

则重新启动, 并修改 Linux 启动参数中 `root` 为 (仅仅是举例, 具体系统的启动命令行很可能会有改变):

```
param set linux_cmd_line "noinitrd root=/dev/mtdblock3 init=/linuxrc console=ttyS0"
```

之后就可以直接以 `/dev/mtdblock3` 中的根文件系统启动了。

YAFFS2 的移植方法与 YAFFS 类似, 而且, 目前我们已经没有必要再移植 YAFFS 了, 因为 YAFFS2 的源码直接包含了对 YAFFS 的支持。LDD6410 开发板对 NAND 采用了 YAFFS2 文件系统, 其内核已完整支持 YAFFS2。

YAFFS 源代码的下载地址为:

```
http://www.aleph1.co.uk/cgi-bin/viewcvs.cgi/yaffs.tar.gz?view=tar
```

YAFFS2 源代码的下载地址为:

```
http://www.aleph1.co.uk/cgi-bin/viewcvs.cgi/yaffs2.tar.gz?view=tar
```

YAFFS2 源代码包的 `utils` 目录下包含了 `mkyaffsimage`、`mkyaffs2image` 工具的源代码, 编译即可生成 `mkyaffsimage`、`mkyaffs2image` 工具。YAFFS 不支持大页的 NAND Flash, 一般只用于页大小为 512Byte 的 NAND Flash, 对于页大小为 2KB 的 NAND Flash 而言, 只能使用 YAFFS2。

运行 “`mkyaffsimage dir imagename`” 可以制作出 YAFFS 文件系统的镜像。需要注意的是, 使用 `mkyaffsimage` 制作出来的 YAFFS 映像文件与通常的文件系统的映像文件不同, 因为在 `image` 文件里除了以 512 字节为单位的一个页的数据外, 同时紧跟在后还包括了 16 字节为单位的 NAND OOB 数据, 因此, YAFFS 映像的下载工具必须将映像中的额外数据写入到 NAND 的 OOB 中。

`nandwrite` 工具“名义上”可以支持 YAFFS 映像的烧录, 但是考虑到实际上 NAND ECC 模式和分布的不确定性, 而 `mkyaffsimage` 生成的映像采用固定的 OOB 区域分布, 因此, 为了完全支持自适应的 YAFFS 映像烧录, 必须在烧录工具中先解析 NAND 驱动的 OOB layout, 再将记录在 YAFFS 映像中的 OOB 数据转换为适合于在系统中 NAND 存放的形式。

对于各种文件系统而言, 如果想在启动过程中将 Flash 的 `xxxf`s 文件系统分区挂载到某个目录, 可以通过修改启动的 `rc` 脚本或 `/etc/fstab` 来完成, 需在启动 `rc` 脚本中增加 “`mount -t xxxfs /dev/mtdblock3 /mnt/Flash0`” 类似语句, 或在 `fstab` 中增加 “`/dev/mtdblock3 /mnt/Flash0 xxxfs defaults 0 0`” 类似语句。

19.6.5 UBI/UBIFS

UBIFS 是由 Thomas Gleixner, Artem Bityutskiy 等人于 2006 年发起, 致力于开发性能卓越、扩展性高的 Flash 专用文件系统。UBI (unsorted block images) 是一种类似于 LVM 的逻辑卷管理层, 主要实现损益均衡, 逻辑擦除块、卷管理和坏块管理等, 而 UBIFS 则是基于 UBI 的 Flash 日志文件系统。UBIFS 并不直接工作于 MTD 之上而是工作于 UBI 卷之上, 这是它与 JFFS2、YAFFS2 的一个显著区别。

为了使用 UBIFS, 我们需要在配置内核时使能如下选项:

```
Device Drivers  --->
  Memory Technology Device (MTD) support  --->
    UBI - Unsorted block images  --->
      <*> Enable UBI
      <*> MTD devices emulation driver (gluebi) (NEW)
File systems  --->
  Miscellaneous filesystems  --->
    <*> UBIFS file system support
```

下面给出一个使用制作、烧录和使用 UBIFS 的过程的例子。

(1) 在 PC 上通过 mtd-utils 制作 UBI 映像:

```
mkfs.ubifs -r rootfs -m 2048 -e 129024 -c 4096 -o ubifs.img
ubinize -o ubi.img -m 2048 -s 512 -p 128KiB ubifs.conf
```

以上命令对应的 Flash 的 page 大小为 2048 字节, subpage 大小为 512 字节, eraseblock 大小为 128KB。rootfs 为要制作的根文件系统的目录。

(2) 在目标机上烧录映像:

```
root:/> ubiformat /dev/mtd1 -s 512 -f ubi.img
ubiformat: mtd1 (NAND), size 130023424 bytes (124.0 MiB), 131072 eraseblocks of 131072
bytes (128.0 KiB), min. I/O size 2048 bytes
libscan: scanning eraseblock 991 -- 100 % complete
ubiformat: 992 eraseblocks are supposedly empty
ubiformat: flashing eraseblock 15 -- 100 % complete
ubiformat: formatting eraseblock 991 -- 100 % complete
```

(3) 通过 ubiattach 关联 MTD UBI:

```
root:/> ubiattach /dev/ubi_ctrl -m 1
UBI: attaching mtd1 to ubi0
UBI: physical eraseblock size: 131072 bytes (128 KiB)
UBI: logical eraseblock size: 129024 bytes
UBI: smallest flash I/O unit: 2048
UBI: sub-page size: 512
UBI: VID header offset: 512 (aligned 512)
UBI: data offset: 2048
UBI: volume 0 ("rootfs") re-sized from 17 to 979 LEBs
UBI: attached mtd1 to ubi0
UBI: MTD device name: "file system(nand)"
UBI: MTD device size: 124 MiB
UBI: number of good PEBs: 992
UBI: number of bad PEBs: 0
UBI: max. allowed volumes: 128
UBI: wear-leveling threshold: 4096
UBI: number of internal volumes: 1
UBI: number of user volumes: 1
UBI: available PEBs: 0
```



```
UBI: total number of reserved PEBs: 992
UBI: number of PEBs reserved for bad PEB handling: 9
UBI: max/mean erase counter: 0/0
UBI: image sequence number: 0
UBI: background thread "ubi_bgt0d" started, PID 179
UBI device number 0, total 992 LEBs (127991808 bytes, 122.1 MiB), available 0 LEBs (0
bytes), LEB size 129024 bytes (126.0 KiB)
```

(4) 挂载 UBIFS:

```
root:/> mount -t ubifs ubi0:rootfs /mnt
UBIFS: mounted UBI device 0, volume 0, name "rootfs"
UBIFS: file system size: 124895232 bytes (121968 KiB, 119 MiB, 968 LEBs)
UBIFS: journal size: 9033728 bytes (8822 KiB, 8 MiB, 71 LEBs)
UBIFS: media format: w4/r0 (latest is w4/r0)
UBIFS: default compressor: lzo
UBIFS: reserved for root: 0 bytes (0 KiB)
```

(5) 现在我们可以通过 mount 和 ubinfo 命令查看下结果:

```
root:/> mount
rootfs on / type rootfs (rw)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
...
ubi0:rootfs on /mnt type ubifs (rw,relatime)

root:/mnt> ubinfo
UBI version: 1
Count of UBI devices: 1
UBI control device major/minor: 10:63
Present UBI devices: ubi0
```

UBIFS 被认为是下一代的 JFFS2, 它也支持运行时压缩, 但是挂载比 JFFS2 快。另外, 被用于 NAND 时, 其设计以及性能都优越于 YAFFS2, 特别是工作在大页 MLC NAND Flash 上面。因此, 目前许多项目中都正在使用 UBIFS 替代 YAFFS2。

19.7 总结

本章主要讲解了 Linux 系统中 MTD 系统的层次和接口, NOR 和 NAND Flash 驱动的设计方法及如何在其上建立 Flash 文件系统。

由于引入了 MTD 系统以及 MTD 下层的通用 NOR 和 NAND 驱动, Linux 中 NOR 和 NAND Flash 芯片级驱动的设计难度被大大降低。尤其对于 NOR 而言, drivers/mtd/maps/physmap.c 驱动支持了绝大不多情况下的 NOR 驱动, 使得移植 NOR 驱动的工作仅仅只需要在 BSP 中添加相关的 platform 信息。

在串口驱动部分, 本章讲解了 tty_driver 到 uart_driver 的角色转换, 在 Flash 驱动中, 本章讲解了 mtd_info 向 map_info/nand_chip 的转移, 可以说, Linux 驱动的这种分层设计思想是贯穿各种 Linux 驱动框架始终的。

LINUX

第20章

USB 主机与设备驱动

在 Linux 系统中，提供了主机侧和设备侧视角的 USB 驱动框架，本章主要讲解从主机侧角度看到的 USB 主机控制器驱动和设备驱动，以及从设备侧角度看到的设备控制器和 gadget 驱动。

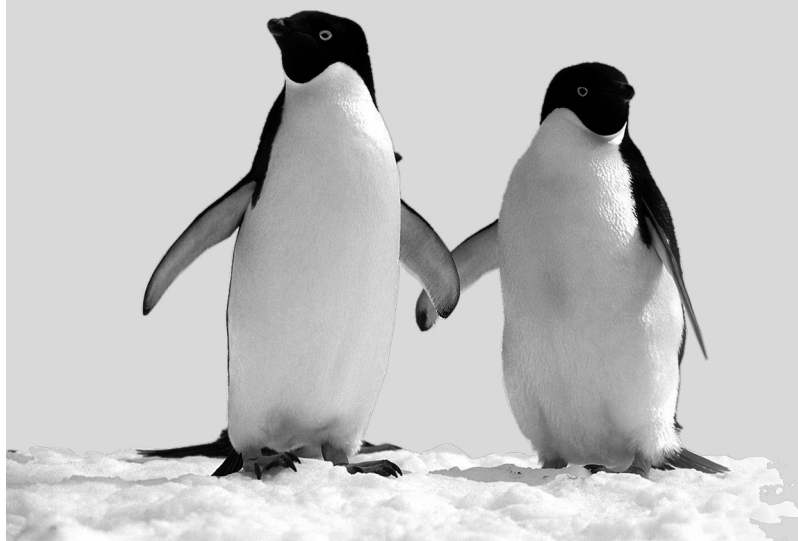
20.1 节给出了 Linux 系统中 USB 驱动的整体视图，讲解了 Linux 中主机侧和设备侧角度的 USB 驱动层次。

从主机侧的角度而言，需要编写的 USB 驱动程序包括主机控制器驱动和设备驱动两类，USB 主机控制器驱动程序控制插入其中的 USB 设备，而 USB 设备驱动程序控制该设备如何作为从设备与主机通信。本章 20.2 节分析了 USB 主机控制器驱动的结构并给出 LDD6410 USB 1.1 主机实例，20.3 节讲解了 USB 设备驱动的结构及其设备请求块处理过程并给出了 USB 键盘驱动作为实例。

从设备侧的角度而言，包含编写 USB 设备控制器（UDC）驱动和 gadget 驱动两类，20.4 对 UDC 和 gadget 驱动进行了讲解，并给出了 LDD6410 UDC 和 file storage gadget 作为实例。

20.5 节简单地介绍了一下 USB OTG 驱动。

20.1 节与 20.2~20.5 节是整体与部分的关系。





20.1 Linux USB 驱动层次

20.1.1 主机侧与设备侧 USB 驱动

USB 采用树形拓扑结构，主机侧和设备侧的 USB 控制器分别称为主机控制器（Host Controller）和 USB 设备控制器（UDC），每条总线上只有一个主机控制器，负责协调主机和设备间的通信，而设备不能主动向主机发送任何消息。如图 20.1 所示，在 Linux 系统中，USB 驱动可以从两个角度去观察，一个角度是主机侧，一个角度是设备侧。

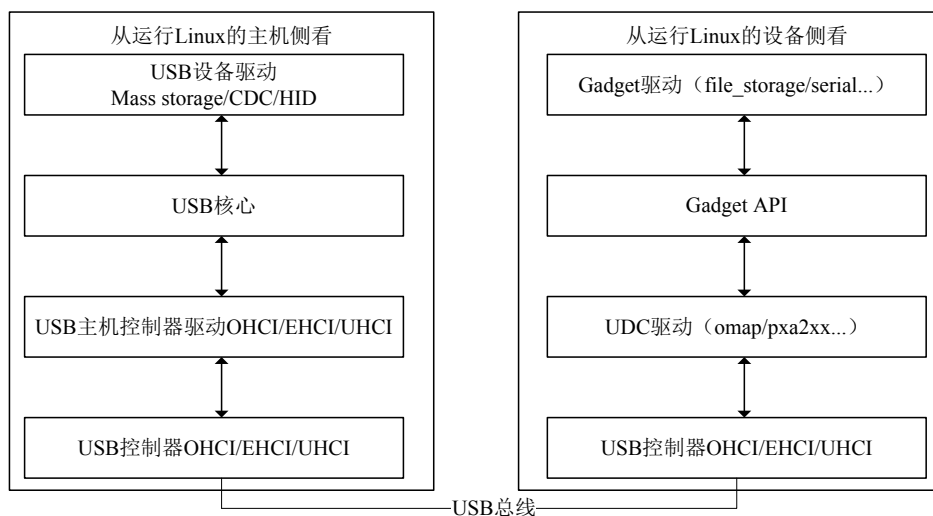


图 20.1 Linux USB 驱动总体结构

如图 20.1 的左侧所示，从主机侧的观念去看，在 Linux 驱动中，USB 驱动处于最底层的是 USB 主机控制器硬件，在其之上运行的是 USB 主机控制器驱动，主机控制器之上为 USB 核心层，再上层为 USB 设备驱动层（插入主机上的 U 盘、鼠标、USB 转串口等设备驱动）。因此，在主机侧的层次结构中，要实现的 USB 驱动包括两类：USB 主机控制器驱动和 USB 设备驱动，前者控制插入其中的 USB 设备，后者控制 USB 设备如何与主机通信。Linux 内核 USB 核心负责 USB 驱动管理和协议处理的主要工作。主机控制器驱动和设备驱动之间的 USB 核心非常重要，其功能包括：通过定义一些数据结构、宏和功能函数，向上为设备驱动提供编程接口，向下为 USB 主机控制器驱动提供编程接口；通过全局变量维护整个系统的 USB 设备信息；完成设备热插拔控制、总线数据传输控制等。

如图 20.1 的右侧所示，Linux 内核中 USB 设备侧驱动程序分为 3 个层次：UDC 驱动程序、Gadget API 和 Gadget 驱动程序。UDC 驱动程序直接访问硬件，控制 USB 设备和主机间的底层通信，向上层提供与硬件相关操作的回调函数。当前 Gadget API 是 UDC 驱动程序回调函数的简单包装。Gadget 驱动程序具体控制 USB 设备功能的实现，使设备表现出“网络连接”、“打印机”

或“USB Mass Storage”等特性，它使用 Gadget API 控制 UDC 实现上述功能。Gadget API 把下层的 UDC 驱动程序和上层的 Gadget 驱动程序隔离开，使得在 Linux 系统中编写 USB 设备侧驱动程序时能够把功能的实现和底层通信分离。

20.1.2 设备、配置、接口、端点

在 USB 设备的逻辑组织中，包含设备、配置、接口和端点 4 个层次。

每个 USB 设备都提供了不同级别的配置信息，可以包含一个或多个配置，不同的配置使设备表现出不同的功能组合（在探测/连接期间需从其中选定一个），配置由多个接口组成。

在 USB 协议中，接口由多个端点组成，代表一个基本的功能，是 USB 设备驱动程序控制的对象，一个功能复杂的 USB 设备可以具有多个接口。每个配置中可以有多个接口，而设备接口是端点的汇集（collection）。例如，USB 扬声器可以包含一个音频接口以及对旋钮和按钮的接口。一个配置中的所有接口可以同时有效，并可被不同的驱动程序连接。每个接口可以有备用接口，以提供不同质量的服务参数。

端点是 USB 通信的最基本形式，每一个 USB 设备接口在主机看来就是一个端点的集合。主机只能通过端点与设备进行通信，以使用设备的功能。在 USB 系统中每一个端点都有惟一的地址，这是由设备地址和端点号给出的。每个端点都有一定的属性，其中包括传输方式、总线访问频率、带宽、端点号和数据包的最大容量等。一个 USB 端点只能在一个方向承载数据，或者从主机到设备（称为输出端点），或者从设备到主机（称为输入端点），因此端点可看作一个单向的管道。端点 0 通常为控制端点，用于设备初始化参数等。只要设备连接到 USB 上并且上电端点 0 就可以被访问。端点 1、2 等一般用作数据端点，存放主机与设备间往来的数据。

总体而言，USB 设备非常复杂，由许多不同的逻辑单元组成，如图 20.2 所示，这些单元之间的关系如下：

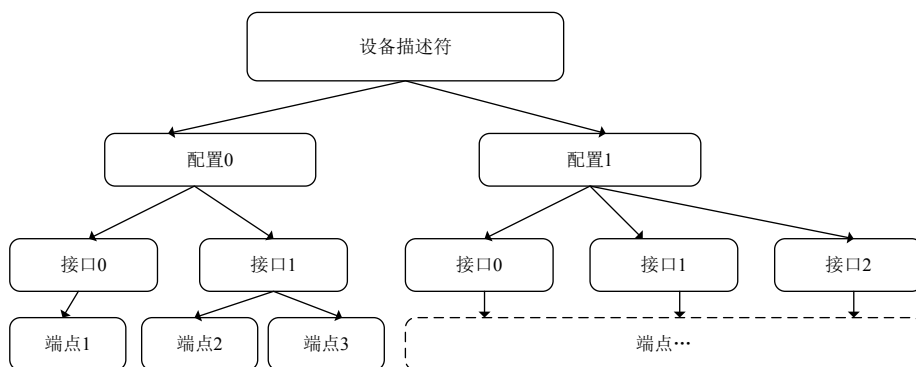


图 20.2 USB 设备、配置、接口和端点

- 设备通常有一个或多个配置；
- 配置通常有一个或多个接口；
- 接口通常有一个或多个设置；
- 接口有零或多个端点。

这种层次化配置信息在设备中通过一组标准的描述符来描述，如下所示。



- 设备描述符：关于设备的通用信息，如供应商 ID、产品 ID 和修订 ID，支持的设备类、子类和适用的协议以及默认端点的最大包大小等。在 Linux 内核中，USB 设备用 `usb_device` 结构体来描述，USB 设备描述符定义为 `usb_device_descriptor` 结构体，如代码清单 20.1 所示。

代码清单 20.1 `usb_device_descriptor` 结构体

```
1 struct usb_device_descriptor {
2     __u8  bLength; /* 描述符长度 */
3     __u8  bDescriptorType; /* 描述符类型编号 */
4
5     __le16 bcdUSB; /* USB 版本号 */
6     __u8  bDeviceClass; /* USB 分配的设备类 code */
7     __u8  bDeviceSubClass; /* USB 分配的子类 code */
8     __u8  bDeviceProtocol; /* USB 分配的协议 code */
9     __u8  bMaxPacketSize0; /* endpoint0 最大包大小 */
10    __le16 idVendor; /* 厂商编号 */
11    __le16 idProduct; /* 产品编号 */
12    __le16 bcdDevice; /* 设备出厂编号 */
13    __u8  iManufacturer; /* 描述厂商字符串的索引 */
14    __u8  iProduct; /* 描述产品字符串的索引 */
15    __u8  iSerialNumber; /* 描述设备序列号字符串的索引 */
16    __u8  bNumConfigurations; /* 可能的配置数量 */
17 } __attribute__((packed));
```

- 配置描述符：此配置中的接口数、支持的挂起和恢复能力以及功率要求。USB 配置在内核中使用 `usb_host_config` 结构体描述，USB 配置描述符定义为结构体 `usb_config_descriptor`，如代码清单 20.2 所示。

代码清单 20.2 `usb_config_descriptor` 结构体

```
1 struct usb_config_descriptor {
3     __u8  bLength; /* 描述符长度 */
4     __u8  bDescriptorType; /* 描述符类型编号 */
5
6     __le16 wTotalLength; /* 配置所返回的所有数据的大小 */
7     __u8  bNumInterfaces; /* 配置所支持的接口数 */
8     __u8  bConfigurationValue; /* Set_Configuration 命令需要的参数值 */
9     __u8  iConfiguration; /* 描述该配置的字符串的索引值 */
10    __u8  bmAttributes; /* 供电模式的选择 */
11    __u8  bMaxPower; /* 设备从总线提取的最大电流 */
12 } __attribute__((packed));
```

- 接口描述符：接口类、子类和适用的协议，接口备用配置的数目和端点数目。USB 接口在内核中使用 `usb_interface` 结构体描述，USB 接口描述符定义为结构体 `usb_interface_descriptor`，如代码清单 20.3 所示。

代码清单 20.3 `usb_interface_descriptor` 结构体

```
1 struct usb_interface_descriptor {
3     __u8  bLength; /* 描述符长度 */
4     __u8  bDescriptorType; /* 描述符类型 */
5
6     __u8  bInterfaceNumber; /* 接口的编号 */
7     __u8  bAlternateSetting; /* 备用的接口描述符编号 */
```



```

8  _u8 bNumEndpoints; /* 该接口使用的端点数, 不包括端点 0 */
9  _u8 bInterfaceClass; /* 接口类型 */
10 _u8 bInterfaceSubClass; /* 接口子类型 */
11 _u8 bInterfaceProtocol; /* 接口所遵循的协议 */
12 _u8 iInterface; /* 描述该接口的字符串索引值 */
13 } __attribute__((packed));

```

- 端点描述符：端点地址、方向和类型，支持的最大包大小，如果是中断类型的端点则还包括轮询频率。在 Linux 内核中，USB 端点使用 `usb_host_endpoint` 结构体来描述，USB 端点描述符定义为 `usb_endpoint_descriptor` 结构体，如代码清单 20.4 所示。

代码清单 20.4 `usb_endpoint_descriptor` 结构体

```

1 struct usb_endpoint_descriptor {
3  _u8 bLength; /* 描述符长度 */
4  _u8 bDescriptorType; /* 描述符类型 */
5  _u8 bEndpointAddress; /* 端点地址: 0~3 位是端点号, 第 7 位是方向 (0-OUT, 1-IN) */
6  _u8 bmAttributes; /* 端点属性: bit[0:1] 的值为 00 表示控制, 为 01 表示同步, 为 02 表示批量, 为 03 表示中断 */
7  _le16 wMaxPacketSize; /* 本端点接收或发送的最大信息包的大小 */
8  _u8 bInterval; /* 轮询数据传送端点的时间间隔 */
9                      /* 对于批量传送的端点以及控制传送的端点, 此域忽略 */
10                     /* 对于同步传送的端点, 此域必须为 1 */
11                     /* 对于中断传送的端点, 此域值的范围为 1~255 */
12  _u8 bRefresh;
13  _u8 bSynchAddress;
14 } __attribute__((packed));

```

- 字符串描述符：在其他描述符中会为某些字段提供字符串索引，它们可被用来检索描述性字符串，可以以多种语言形式提供。字符串描述符是可选的，有的设备有，有的设备没有，字符串描述符对应于 `usb_string_descriptor` 结构体，如代码清单 20.5 所示。

代码清单 20.5 `usb_string_descriptor` 结构体

```

1 struct usb_string_descriptor {
3  _u8 bLength; /* 描述符长度 */
4  _u8 bDescriptorType; /* 描述符类型 */
5
6  _le16 wData[1]; /* 以 UTF-16LE 编码 */
7 } __attribute__((packed));

```

例如，笔者在 PC 上插入一个 SanDisk U 盘后，通过 `lsusb` 命令得到这个 U 盘相关的描述符，从中可以显示这个 U 盘包含了一个设备描述符、一个配置描述符、一个接口描述符以及批量输入和批量输出两个端点描述符。呈现出来的信息内容直接对应于 `usb_device_descriptor`、`usb_config_descriptor`、`usb_interface_descriptor`、`usb_endpoint_descriptor`、`usb_string_descriptor` 结构体，如下所示：

```

Bus 001 Device 004: ID 0781:5151 SanDisk Corp.
Device Descriptor:
  bLength                18
  bDescriptorType         1
  bcdUSB                  2.00
  bDeviceClass            0 Interface
  bDeviceSubClass         0
  bDeviceProtocol         0
  bMaxPacketSize0        64

```



```
idVendor      0x0781 SanDisk Corp.
idProduct     0x5151
bcdDevice      0.10
iManufacturer  1 SanDisk Corporation
iProduct       2 Cruzer Micro
iSerial        3 20060877500A1BE1FDE1
bNumConfigurations 1
```

Configuration Descriptor:

```
bLength      9
bDescriptorType 2
wTotalLength 32
bNumInterfaces 1
bConfigurationValue 1
iConfiguration 0
bmAttributes  0x80
MaxPower      200mA
```

Interface Descriptor:

```
bLength      9
bDescriptorType 4
bInterfaceNumber 0
bAlternateSetting 0
bNumEndpoints 2
bInterfaceClass 8 Mass Storage
bInterfaceSubClass 6 SCSI
bInterfaceProtocol 80 Bulk (Zip)
iInterface    0
```

Endpoint Descriptor:

```
bLength      7
bDescriptorType 5
bEndpointAddress 0x81 EP 1 IN
bmAttributes  2
    Transfer Type      Bulk
    Synch Type         none
wMaxPacketSize 512
bInterval     0
```

Endpoint Descriptor:

```
bLength      7
bDescriptorType 5
bEndpointAddress 0x01 EP 1 OUT
bmAttributes  2
    Transfer Type      Bulk
    Synch Type         none
wMaxPacketSize 512
bInterval     1
```

```
Language IDs: (length=4)
0409 English(US)
```

20.2 USB 主机控制器驱动

20.2.1 USB 主机控制器驱动的整体结构

USB 主机控制器有 3 种规格: OHCI (Open Host Controller Interface)、UHCI (Universal Host Controller Interface) 和 EHCI (Enhanced Host Controller Interface)。OHCI 驱动程序用来为非 PC 系

统上以及带有 SiS 和 ALi 芯片组的 PC 主板上的 USB 芯片提供支持。UHCI 驱动程序多用来为大多数其他 PC 主板（包括 Intel 和 Via）上的 USB 芯片提供支持。EHCI 由 USB 2.0 规范所提出，它兼容于 OHCI 和 UHCI。UHCI 的硬件线路比 OHCI 简单，所以成本较低，但需要较复杂的驱动程序，CPU 负荷稍重。本节将重点介绍嵌入式系统中常用的 OHCI 主机控制器驱动。

1. 主机控制器驱动

在 Linux 内核中，用 `usb_hcd` 结构体描述 USB 主机控制器驱动，它包含 USB 主机控制器的“家务”信息、硬件资源、状态描述和用于操作主机控制器的 `hc_driver` 等，其定义如代码清单 20.6 所示。

代码清单 20.6 `usb_hcd` 结构体

```

1 struct usb_hcd {
2     /*
3      * housekeeping
4      */
5     struct usb_bus      self;          /* hcd 是一个 bus */
6     struct kref          kref;
7
8     const char          *product_desc; /* 产品/厂商字符串 */
9     char                irq_descr[24]; /* driver + bus # */
10
11     struct timer_list    rh_timer;      /* 驱动根 hub 的 polling */
12     struct urb           *status_urb;   /* 目前的状态 urb */
13 #ifdef CONFIG_PM
14     struct work_struct    wakeup_work; /* 用于远程唤醒 */
15 #endif
16
17     /*
18      * 硬件信息/状态
19      */
20     const struct hc_driver *driver; /* 硬件特定的钩子函数 */
21
22     /* 需要被自动操作的标志 */
23     unsigned long        flags;
24 #define HCD_FLAG_HW_ACCESSIBLE 0x00000001
25 #define HCD_FLAG_SAW_IRQ      0x00000002
26
27     unsigned             rh_registered; /* 根 Hub 已被注册? */
28
29     /* 下一个标志的采用只是“权益之计”，当所有 HCDs 支持新的根 Hub 轮询机制后将移除 */
30
31     unsigned             uses_new_polling;
32     unsigned             poll_rh;       /* 轮询根 Hub 状态? */
33     unsigned             poll_pending;  /* 状态改变了吗? */
34     unsigned             wireless;      /* 无线 USB HCD? */
35     unsigned             authorized_default;
36     unsigned             has_tt;        /* 根 hub 集成了 TT? */
37
38     int                  irq;           /* 分配的中断号 */
39     void __iomem          *regs;         /* 设备内存或 I/O */
40     u64                   rsrc_start;    /* 内存或 I/O 资源开始位置 */
41     u64                   rsrc_len;      /* 内存或 I/O 资源大小 */
42     unsigned              power_budget; /* in mA, 0 = no limit */

```



```
43
44 #define HCD_BUFFER_POOLS      4
45 struct dma_pool                *pool [HCD_BUFFER_POOLS];
46
47 int                            state;
48
49 /* 主机控制器驱动的私有数据 */
50 unsigned long hcd_priv[0]
51     __attribute__((aligned(sizeof(unsigned long))));
52 };
```

usb_hcd 中的 hc_driver 成员非常重要, 它包含具体的用于操作主机控制器的钩子函数, 其定义如代码清单 20.7 所示。

代码清单 20.7 hc_driver 结构体

```
1 struct hc_driver {
2     const char *description; /* "ehci-hcd" 等 */
3     const char *product_desc; /* 产品/厂商字符串 */
4     size_t hcd_priv_size; /* 私有数据的大小 */
5
6     /* 中断处理函数 */
7     irqreturn_t (*irq) (struct usb_hcd *hcd);
8
9     int flags;
10    #define HCD_MEMORY      0x0001    /* HC 寄存器使用的内存和 I/O */
11    #define HCD_USB11      0x0010    /* USB 1.1 */
12    #define HCD_USB2      0x0020    /* USB 2.0 */
13
14    /* 被调用以初始化 HCD 和根 Hub */
15    int (*reset) (struct usb_hcd *hcd);
16    int (*start) (struct usb_hcd *hcd);
17
18    /* 挂起 Hub 后, 进入 D3 (etc) 前被调用 */
19    int (*pci_suspend) (struct usb_hcd *hcd, pm_message_t message);
20
21    /* 在进入 D0 (etc) 后, 恢复 Hub 前调用 */
22    int (*pci_resume) (struct usb_hcd *hcd);
23
24    /* 使 HCD 停止写内存和进行 I/O 操作 */
25    void (*stop) (struct usb_hcd *hcd);
26
27    /* 关闭 HCD */
28    void (*shutdown) (struct usb_hcd *hcd);
29
30    /* 返回目前的帧数 */
31    int (*get_frame_number) (struct usb_hcd *hcd);
32
33    /* 管理 I/O 请求和设备状态 */
34    int (*urb_enqueue) (struct usb_hcd *hcd, struct urb *urb, gfp_t mem_flags);
35    int (*urb_dequeue) (struct usb_hcd *hcd, struct urb *urb, int status);
36
37    /* 释放 endpoint 资源 */
38    void (*endpoint_disable) (struct usb_hcd *hcd, struct usb_host_endpoint *ep);
39
40    /* 根 Hub 支持 */
```

```

41 int(*hub_status_data)(struct usb_hcd *hcd, char *buf);
42 int(*hub_control)(struct usb_hcd *hcd, u16 typeReq, u16 wValue, u16 wIndex,
43     char *buf, u16 wLength);
44 int(*bus_suspend)(struct usb_hcd*);
45 int(*bus_resume)(struct usb_hcd*);
46 int(*start_port_reset)(struct usb_hcd *, unsigned port_num);
47 void (*relinquish_port)(struct usb_hcd *, int);
48 int (*port_handed_over)(struct usb_hcd *, int);
49 };

```

在 Linux 内核中，使用如下函数来创建 HCD:

```

struct usb_hcd *usb_create_hcd (const struct hc_driver *driver,
    struct device *dev, char *bus_name);

```

如下函数被用来增加和移除 HCD:

```

int usb_add_hcd(struct usb_hcd *hcd,
    unsigned int irqnum, unsigned long irqflags);
void usb_remove_hcd(struct usb_hcd *hcd);

```

第 34 行的 `urb_enqueue()` 函数非常关键，实际上，上层通过 `usb_submit_urb()` 提交 1 个 USB 请求后，该函数调用 `usb_hcd_submit_urb()`，并最终调用至 `usb_hcd` 的 `driver` 成员（`hc_driver` 类型）的 `urb_enqueue()`。这里可以先建立一点印象，不理解没有关系，后文会看得更加清楚。

2. OHCI 主机控制器驱动

OHCI HCD 驱动属于 HCD 驱动的实例，它定义了一个 `ohci_hcd` 结构体，作为代码清单 20.6 给出的 `usb_hcd` 结构体的私有数据，这个结构体的定义如代码清单 20.8 所示。

代码清单 20.8 ohci_hcd 结构体

```

1 struct ohci_hcd {
2     spinlock_t lock;
3
4     /* 与主机控制器通信的 I/O 内存 (DMA 一致) */
5     struct ohci_regs _iomem *regs;
6
7     /* 与主机控制器通信的主存 (DMA 一致) */
8     struct ohci_hcca *hcca;
9     dma_addr_t hcca_dma;
10
11     struct ed *ed_rm_list; /* 将被移除 */
12     struct ed *ed_bulk_tail; /* 批量队列尾 */
13     struct ed *ed_control_tail; /* 控制队列尾 */
14     struct ed *periodic[NUM_INTS]; /* int_table “影子” */
15
16     /* OTG 控制器和收发器需要软件交互，其他的外部收发器应该是软件透明的 */
17     struct otg_transceiver *transceiver;
18     void (*start_hnp)(struct ohci_hcd *ohci);
19
20     /* 队列数据的内存管理 */
21     struct dma_pool *td_cache;
22     struct dma_pool *ed_cache;
23     struct td *td_hash[TD_HASH_SIZE];
24     struct list_head pending;
25
26     /* driver 状态 */
27     int num_ports;

```



```

28 int load[NUM_INTS];
29 u32 hc_control; /* 主机控制器控制寄存器的复制 */
30 unsigned long next_statechange; /* 挂起/恢复 */
31 u32 fmininterval; /* 被保存的寄存器 */
32 unsigned autostop:1;
33 unsigned long flags;
34 struct work_struct      nec_work;
35 struct timer_list       unlink_watchdog;
36 unsigned               eds_scheduled;
37 struct ed               *ed_to_check;
38 unsigned               zf_delay;
39 };

```

使用如下内联函数可实现 `usb_hcd` 和 `ohci_hcd` 的相互转换:

```

struct ohci_hcd *hcd_to_ohci (struct usb_hcd *hcd);
struct usb_hcd *ohci_to_hcd (const struct ohci_hcd *ohci);

```

从 `usb_hcd` 得到 `ohci_hcd` 只是取得“私有”数据, 而从 `ohci_hcd` 得到 `usb_hcd` 则是通过 `container_of()` 从结构体成员获得结构体指针。

使用如下函数可初始化 OHCI 主机控制器:

```
int ohci_init (struct ohci_hcd *ohci);
```

如下函数分别用于开启、停止及复位 OHCI 控制器:

```

int ohci_run (struct ohci_hcd *ohci);
void ohci_stop (struct usb_hcd *hcd);
void ohci_usb_reset (struct ohci_hcd *ohci);

```

OHCI 主机控制器驱动的主机工作仍然是实现代码清单 20.7 给出的 `hc_driver` 结构体中的成员函数。

20.2.2 实例：S3C6410 USB 1.1 主机驱动

S3C6410 内部集成了 2 个 USB 控制器, 1 个是 USB 1.1 主机控制器, 支持低速和全速 USB 设备, 另外 1 个是 USB 2.0 支持 OTG 的 USB 控制器。

S3C6410 的 USB1.1 主机控制器驱动由 `drivers/usb/host/ohci-hcd.c` (服务于各种 SoC 情况下的 OHCI 主机驱动通用部分) 和 `drivers/usb/host/ohci-s3c2410.c` (服务于 S3C2410、S3C64XX 和 S5PC1XX) 文件共同完成, 前者通过“`#include "ohci-s3c2410.c"`”语句包含了后者, 详细情况如下:

```

#if defined(CONFIG_ARCH_S3C2410) || defined(CONFIG_ARCH_S3C64XX) || defined(CONFIG_ARCH_S5PC1XX)
#include "ohci-s3c2410.c"
#define PLATFORM_DRIVER      ohci_hcd_s3c2410_driver
#endif

```

`drivers/usb/host/ohci-hcd.c` 只是根据定义的 CPU 体系结构引用对应的 `platform_driver` 并注册之, 对于 S3C6410 为 `ohci_hcd_s3c2410_driver`。

S3C6410 USB1.1 主机控制器驱动的 `hc_driver` 结构体中的大多数成员函数都是通用的 `ohci_xxx()` 函数, 而 `start()`、`hub_status_data()`、`hub_control()` 函数则针对 S3C6410 而编写的, 如代码清单 20.9 所示。

代码清单 20.9 S3C2410 主机控制器驱动的 `hc_driver` 结构体

```

1 static const struct hc_driver ohci_s3c2410_hc_driver = {
2     .description =      hcd_name,
3     .product_desc =     "S3C24XX OHCI",

```

```

4  .hcd_priv_size =  sizeof(struct ohci_hcd),
5
6  /* 通用硬件联接 */
7  .irq =  ohci_irq,
8  .flags = HCD_USB11 | HCD_MEMORY, /* USB 1.1 标准, hc 寄存器位于内存 */
9
10 /* 基本的生命周期操作 */
11 .start =  ohci_s3c2410_start,
12 .stop =  ohci_stop,
13
14 /* 管理 I/O 请求和相关的设备资源 */
15 .urb_enqueue =  ohci_urb_enqueue,
16 .urb_dequeue =  ohci_urb_dequeue,
17 .endpoint_disable =  ohci_endpoint_disable,
18
19 /* 调度支持 */
20 .get_frame_number =  ohci_get_frame,
21
22 /* 根 Hub 支持 */
23 .hub_status_data =  ohci_s3c2410_hub_status_data,
24 .hub_control =  ohci_s3c2410_hub_control,
25 #ifdef CONFIG_PM
26 .bus_suspend =  ohci_bus_suspend,
27 .bus_resume =  ohci_bus_resume,
28 #endif
29 .start_port_reset =  ohci_start_port_reset,
30 };

```

hc_driver 的 start()成员函数用于初始化 OHCI 并启动主机控制器，如代码清单 20.10 所示。

代码清单 20.10 S3C6410 USB1.1 主机控制器驱动的 start()函数

```

1 static int ohci_s3c2410_start (struct usb_hcd *hcd)
2 {
3     struct ohci_hcd  *ohci = hcd_to_ohci (hcd);
4     int ret;
5
6     if ((ret = ohci_init(ohci)) < 0) /* 初始化 ohci_hcd */
7         return ret;
8
9     if ((ret = ohci_run (ohci)) < 0) { /* 启动 ohci_hcd */
10         err ("can't start %s", hcd->self.bus_name);
11         ohci_stop (hcd);
12         return ret;
13     }
14
15     return 0;
16 }

```

hc_driver 的 hub_control()成员函数 ohci_s3c2410_hub_control()中的主体是调用通用的 ohci_hub_control()函数，hub_status_data()成员函数 ohci_s3c2410_hub_status_data()的主体是调用通用的 ohci_hub_status_data()函数。

LDD6410 开发板 USB 1.1 主机接口原理如图 20.3，对于 LDD6410 开发板而言，在内核中开



启“OHCI HCD support”、“USB Human Interface Device (full HID) support”和“USB Mass Storage support”选项可以使我们支持 USB 1.1 主机、U 盘以及 HID 设备。

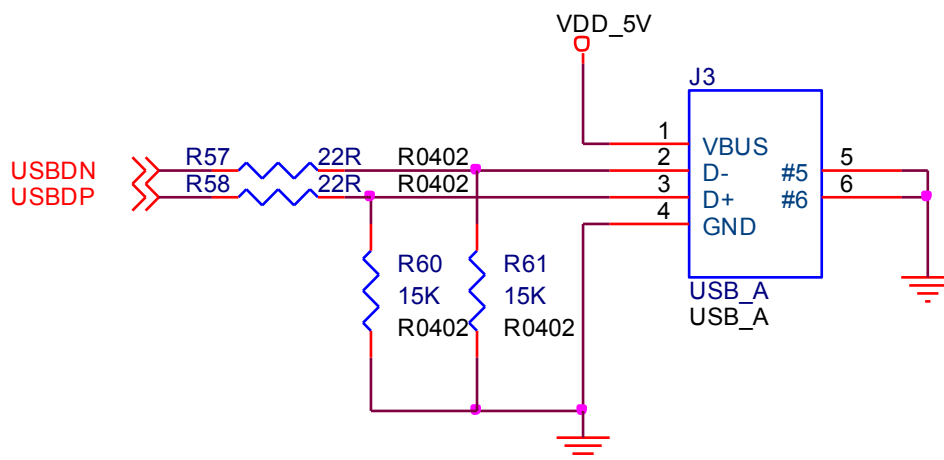


图 20.3 LDD6410 USB 1.1 主机接口原理图

当插入一个 USB 鼠标，控制台会打印类似信息：

```
usb 11: configuration #1 chosen from 1 choice input: USB Mouse as /class/input/input2
genericusb 0003:1267:0201.0001: input: USB HID v1.00 Mouse [USB Mouse] on usb
s3c24xx1/input0
```

拔出 USB 鼠标后，控制台打印：

```
usb 11: USB disconnect, address 2
```

插入一个 U 盘，控制台会打印类似信息：

```
usb 11: new full speed USB device using s3c2410ohci and address 4
usb 11: configuration #1 chosen from 1 choice
scsi0 : SCSI emulation for USB Mass Storage devices
# scsi 0:0:0:0: DirectAccess  Lenovo  USB Flash Drive  1100 PQ: 0 ANSI: 0 CCS
sd 0:0:0:0: [sda] 7831552 512byte hardware sectors: (4.00 GB/3.73 GiB)
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] Assuming drive cache: write through
sd 0:0:0:0: [sda] 7831552 512byte hardware sectors: (4.00 GB/3.73 GiB)
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] Assuming drive cache: write through sda: sda1
sd 0:0:0:0: [sda] Attached SCSI removable disk sd 0:0:0:0: Attached scsi generic sg0
type 0
```

通过 mount 命令来挂载这个 U 盘：

```
# mount /dev/sda1 t vfat /mnt
```

20.3 USB 设备驱动

20.3.1 USB 设备驱动整体结构

这里所说的 USB 设备驱动指的是从主机角度观察，怎样访问被插入的 USB 设备，而不是

指 USB 设备内部本身运行的固件程序。Linux 系统实现了几类通用的 USB 设备驱动（也称客户驱动），划分为如下几个设备类。

- 音频设备类。
- 通信设备类。
- HID（人机接口）设备类。
- 显示设备类。
- 海量存储设备类。
- 电源设备类。
- 打印设备类。
- 集线器设备类。

一般的通用的 Linux 设备（如 U 盘、USB 鼠标、USB 键盘等）都不需要工程师再编写驱动，需要编写的是特定厂商、特定芯片的驱动，而且往往也可以参考内核中已提供的驱动的模板。

Linux 内核为各类 USB 设备分配了相应的设备号，如 ACM USB 调制解调器的主设备号为 166（默认设备名/dev/ttyACMn）、USB 打印机的主设备号为 180，次设备号为 0~15（默认设备名/dev/lpn）、USB 串口的主设备号为 188（默认设备名/dev/ttyUSBn）等，详见 <http://www.lanana.org/> 网站的设备列表。

内核中提供了 USB 设备文件系统（usbdevfs，Linux 2.6 改为 usbfs，即 USB 文件系统），它和/proc 类似，都是动态产生的。通过在/etc/fstab 文件中添加如下一行：

```
none /proc/bus/usb usbfs defaults
```

或者输入命令：

```
mount -t usbfs none /proc/bus/usb
```

可以实现 USB 设备文件系统的挂载。

一个典型的/proc/bus/usb/devices 文件的结构如下（笔者在 PC 上插入了一个 SanDisk U 盘）：

```
T: Bus=01 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#= 1 Spd=12 MxCh= 2
B: Alloc= 0/900 us ( 0%), #Int= 0, #Iso= 0
D: Ver= 1.10 Cls=09(hub ) Sub=00 Prot=00 MxPS=64 #Cfgs= 1
P: Vendor=0000 ProdID=0000 Rev= 2.06
S: Manufacturer=Linux 2.6.15.5 uhci_hcd
S: Product=UHCI Host Controller
S: SerialNumber=0000:00:07.2
C:* #Ifs= 1 Cfg#= 1 Atr=c0 MxPwr= 0mA
I: If#= 0 Alt= 0 #EPs= 1 Cls=09(hub ) Sub=00 Prot=00 Driver=hub
E: Ad=81(I) Atr=03(Int.) MxPS= 2 Iv1=255ms

T: Bus=01 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#= 2 Spd=12 MxCh= 0
D: Ver= 2.00 Cls=00(>ifc ) Sub=00 Prot=00 MxPS=64 #Cfgs= 1
P: Vendor=0781 ProdID=5151 Rev= 0.10
S: Manufacturer=SanDisk Corporation
S: Product=Cruzer Micro
S: SerialNumber=20060877500A1BE1FDE1
C:* #Ifs= 1 Cfg#= 1 Atr=80 MxPwr=200mA
I: If#= 0 Alt= 0 #EPs= 2 Cls=08(stor.) Sub=06 Prot=50 Driver=(none)
E: Ad=81(I) Atr=02(Bulk) MxPS= 512 Iv1=0ms
E: Ad=01(O) Atr=02(Bulk) MxPS= 512 Iv1=0ms
```



通过分析 `usbfs` 中记录的信息,可以得到系统中 USB 完整的信息,例如, `usbview` 可以以图形化的方式显示系统中的 USB 设备。

当然,在编译 Linux 内核时,应该包括“USB device filesystem”。`usbfs` 动态跟踪总线上插入和移除的设备,通过它可以查看系统中 USB 设备的信息,包括拓扑、带宽、设备描述符信息、产品 ID、字符串描述符、配置描述符、接口描述符、端点描述符等。

此外,在 `sysfs` 文件系统中,同样包含了 USB 相关信息的描述,但只限于接口级别。USB 设备和 USB 接口在 `sysfs` 中均表示为单独的 USB 设备,其目录命名规则如下:

根集线器-集线器端口号 (-集线器端口号-...):配置.接口

下面讲解 `/sys/bus/usb` 目录的树形结构实例,其中的多数文件都是到 `/sys/devices` 及 `/sys/drivers` 中相应文件的链接。

```
usb
|-- devices
|   |-- 1-0:1.0 → ../../../../devices/pci0000:00/0000:00:07.2/usb1/1-0:1.0
|   |-- 1-1 → ../../../../devices/pci0000:00/0000:00:07.2/usb1/1-1
|   |-- 1-1:1.0 → ../../../../devices/pci0000:00/0000:00:07.2/usb1/1-1/1-1:1.0
|   '-- usb1 → ../../../../devices/pci0000:00/0000:00:07.2/usb1
'-- drivers
    |-- hub
    |   |-- 1-0:1.0 → ../../../../devices/pci0000:00/0000:00:07.2/usb1/1-0:1.0
    |   |-- bind
    |   |-- module → ../../../../module/usbcore
    |   '-- unbind
    |-- usb
    |   |-- 1-1 → ../../../../devices/pci0000:00/0000:00:07.2/usb1/1-1
    |   |-- bind
    |   |-- module → ../../../../module/usbcore
    |   |-- unbind
    |   '-- usb1 → ../../../../devices/pci0000:00/0000:00:07.2/usb1
    '-- usbfs
        |-- bind
        |-- module → ../../../../module/usbcore
        '-- unbind
```

正如 `tty_driver`、`i2c_driver` 等,在 Linux 内核中,使用 `usb_driver` 结构体描述一个 USB 设备驱动, `usb_driver` 结构体的定义如代码清单 20.11 所示。

代码清单 20.11 `usb_driver` 结构体

```
1 struct usb_driver {
2     const char *name; /* 驱动名称 */
3     int (*probe) (struct usb_interface *intf,
4                   const struct usb_device_id *id); /* 探测函数 */
5     void (*disconnect) (struct usb_interface *intf); /* 断开函数 */
6     int (*ioctl) (struct usb_interface *intf, unsigned int code,
7                   void *buf); /* I/O 控制函数 */
8     int (*suspend) (struct usb_interface *intf, pm_message_t message); /* 挂起函数 */
9     int (*resume) (struct usb_interface *intf); /* 恢复函数 */
10    int (*reset_resume) (struct usb_interface *intf);
11    void (*pre_reset) (struct usb_interface *intf);
12    void (*post_reset) (struct usb_interface *intf);
13    const struct usb_device_id *id_table; /* usb_device_id 表指针 */
14    struct usb_dynids dynids;
15    struct usbdrv_wrap drvwrap;
```

```

16 unsigned int no_dynamic_id:1;
17 unsigned int supports_autosuspend:1;
18 unsigned int soft_unbind:1;
19 };

```

在编写新的 USB 设备驱动时，主要应该完成的工作是 `probe()` 和 `disconnect()` 函数，即探测和断开函数，它们分别在设备被插入和拔出的时候被调用，用于初始化和释放软硬件资源。对 `usb_driver` 的注册和注销通过这两个函数完成：

```

int usb_register(struct usb_driver *new_driver)
void usb_deregister(struct usb_driver *driver);

```

`usb_driver` 结构体中的 `id_table` 成员描述了这个 USB 驱动所支持的 USB 设备列表，它指向一个 `usb_device_id` 数组，`usb_device_id` 结构体用于包含 USB 设备的制造商 ID、产品 ID、产品版本、设备类、接口类等信息及其要匹配标志成员 `match_flags`（标明要与哪些成员匹配，包含 `DEV_LO`、`DEV_HI`、`DEV_CLASS`、`DEV_SUBCLASS`、`DEV_PROTOCOL`、`INT_CLASS`、`INT_SUBCLASS`、`INT_PROTOCOL`）。可以借助下面一组宏来生成 `usb_device_id` 结构体的实例：

```
USB_DEVICE(vendor, product)
```

该宏根据制造商 ID 和产品 ID 生成一个 `usb_device_id` 结构体的实例，在数组中增加该元素将意味着该驱动可支持匹配制造商 ID、产品 ID 的设备。

```
USB_DEVICE_VER(vendor, product, lo, hi)
```

该宏根据制造商 ID、产品 ID、产品版本的最小值和最大值生成一个 `usb_device_id` 结构体的实例，在数组中增加该元素将意味着该驱动可支持匹配制造商 ID、产品 ID 和 `lo~hi` 范围内版本的设备。

```
USB_DEVICE_INFO(class, subclass, protocol)
```

该宏用于创建一个匹配设备指定类型的 `usb_device_id` 结构体实例。

```
USB_INTERFACE_INFO(class, subclass, protocol)
```

该宏用于创建一个匹配接口指定类型的 `usb_device_id` 结构体实例。

代码清单 20.12 所示为两个用于描述某 USB 驱动所支持的 USB 设备的 `usb_device_id` 结构体数组实例。

代码清单 20.12 `usb_device_id` 结构体数组实例

```

1  /* 本驱动支持的 USB 设备列表 */
2
3  /* 实例 1 */
4  static struct usb_device_id id_table [] = {
5      { USB_DEVICE(VENDOR_ID, PRODUCT_ID) },
6      { },
7  };
8  MODULE_DEVICE_TABLE (usb, id_table);
9
10 /* 实例 2 */
11 static struct usb_device_id id_table [] = {
12     { .idVendor = 0x10D2, .match_flags = USB_DEVICE_ID_MATCH_VENDOR, },
13     { },
14 };
15 MODULE_DEVICE_TABLE (usb, id_table);

```

当 USB 核心检测到某个设备的属性和某个驱动程序的 `usb_device_id` 结构体所携带的信息一致时，这个驱动程序的 `probe()` 函数就被执行。拔掉设备或者卸掉驱动模块后，USB 核心就执行 `disconnect()` 函数来响应这个动作。



上述 `usb_driver` 结构体中的函数是 USB 设备驱动中 USB 相关的部分，而 USB 只是一个总线，真正的 USB 设备驱动的主体工作仍然是 USB 设备本身所属类型的驱动，如字符设备、tty 设备、块设备、输入设备等。因此 USB 设备驱动包含其作为总线上挂在设备的驱动和本身所属设备类型的驱动两部分。

与 `platform_driver` 类似，`usb_driver` 起到了“牵线”的作用，即在 `probe()` 里注册相应的字符、tty 等设备，在 `disconnect()` 注销相应的字符、tty 等设备，而原先对设备的注册和注销一般直接发生在模块加载和卸载函数中。

尽管 USB 本身所属设备驱动的结构与其不挂在 USB 总线上时完全相同，但是在访问方式上却发生了很大的变化，例如，对于 USB 接口的字符设备而言，尽管仍然是 `write()`、`read()`、`ioctl()` 这些函数，但是在这些函数中，贯穿始终的是称为 URB 的 USB 请求块。

如图 20.4 所示，在这棵树里，我们把树根比作主机控制器，树叶比作具体的 USB 设备，树干和树枝就是 USB 总线。树叶本身与树枝通过 `usb_driver` 连接，而树叶本身的驱动（读写、控制）则需要通过其树叶设备本身所属类设备驱动来完成。树根和树叶之间的“通信”依靠在树干和树枝里“流淌”的 URB 来完成。



图 20.4 USB 设备驱动结构

由此可见，`usb_driver` 本身只是起到了找到 USB 设备、管理 USB 设备连接和断开的作用，也就是说，它是公司入口处的“打卡机”，可以获得员工（USB 设备）的上/下班情况。树叶和员工一样，可以是研发工程师也可以是销售工程师，而作为 USB 设备的树叶可以是字符树叶、网络树叶或块树叶，因此必须实现相应设备类的驱动。

20.3.2 USB 请求块 (URB)

1. urb 结构体

USB 请求块 (USB request block, urb) 是 USB 设备驱动中用来描述与 USB 设备通信所用的基本载体和核心数据结构, 非常类似于网络设备驱动中的 sk_buff 结构体。

代码清单 20.13 urb 结构体

```

1 struct urb {
2     /* 私有的: 只能由 USB 核心和主机控制器访问的字段 */
3     struct kref kref; /*urb 引用计数 */
4     void *hcpriv; /* 主机控制器私有数据 */
5     atomic_t use_count; /* 并发传输计数 */
6     u8 reject; /* 传输将失败 */
7     int unlink; /* unlink 错误码 */
8
9     /* 公共的: 可以被驱动使用的字段 */
10    struct list_head urb_list; /* 链表头 */
11    struct usb_anchor *anchor;
12    struct usb_device *dev; /* 关联的 USB 设备 */
13    struct usb_host_endpoint *ep;
14    unsigned int pipe; /* 管道信息 */
15    int status; /* URB 的当前状态 */
16    unsigned int transfer_flags; /* URB_SHORT_NOT_OK | ... */
17    void *transfer_buffer; /* 发送数据到设备或从设备接收数据的缓冲区 */
18    dma_addr_t transfer_dma; /* 用来以 DMA 方式向设备传输数据的缓冲区 */
19    int transfer_buffer_length; /* transfer_buffer 或 transfer_dma 指向缓冲区的大小 */
20
21    int actual_length; /* URB 结束后, 发送或接收数据的实际长度 */
22    unsigned char *setup_packet; /* 指向控制 URB 的设置数据包的指针 */
23    dma_addr_t setup_dma; /* 控制 URB 的设置数据包的 DMA 缓冲区 */
24    int start_frame; /* 等时传输中用于设置或返回初始帧 */
25    int number_of_packets; /* 等时传输中等时缓冲区数量 */
26    int interval; /* URB 被轮询到的时间间隔 (对中断和等时 urb 有效) */
27    int error_count; /* 等时传输错误数量 */
28    void *context; /* completion 函数上下文 */
29    usb_complete_t complete; /* 当 URB 被完全传输或发生错误时, 被调用 */
30    /* 单个 URB 一次可定义多个等时传输时, 描述各个等时传输 */
31    struct usb_iso_packet_descriptor iso_frame_desc[0];
32 };

```

2. urb 处理流程

USB 设备中的每个端点都处理一个 urb 队列, 在队列被清空之前, 一个 urb 的典型生命周期如下。

(1) 被一个 USB 设备驱动创建。

创建 urb 结构体的函数为:

```
struct urb *usb_alloc_urb(int iso_packets, int mem_flags);
```

iso_packets 是这个 urb 应当包含的等时数据包的数目, 若为 0 表示不创建等时数据包。mem_flags 参数是分配内存的标志, 和 kmalloc() 函数的分配标志参数含义相同。如果分配成功, 该函数返回一个 urb 结构体指针, 否则返回 0。

urb 结构体在驱动中不能静态创建, 因为这可能破坏 USB 核心给 urb 使用的引用计数方法。



usb_alloc_urb()的“反函数”为:

```
void usb_free_urb(struct urb *urb);
```

该函数用于释放由 usb_alloc_urb()分配的 urb 结构体。

(2) 初始化, 被安排给一个特定 USB 设备的特定端点。

对于中断 urb, 使用 usb_fill_int_urb()函数来初始化 urb, 如下所示:

```
void usb_fill_int_urb(struct urb *urb, struct usb_device *dev,
    unsigned int pipe, void *transfer_buffer,
    int buffer_length, usb_complete_t complete,
    void *context, int interval);
```

urb 参数指向要被初始化的 urb 的指针; dev 指向这个 urb 要被发送到的 USB 设备; pipe 是这个 urb 要被发送到的 USB 设备的特定端点; transfer_buffer 是指向发送数据或接收数据的缓冲区的指针, 和 urb 一样, 它也不能是静态缓冲区, 必须使用 kmalloc()来分配; buffer_length 是 transfer_buffer 指针所指向缓冲区的大小; complete 指针指向当这个 urb 完成时被调用的完成处理函数; context 是完成处理函数的“上下文”; interval 是这个 urb 应当被调度的间隔。

上述函数参数中的 pipe 使用 usb_sndintpipe()或 usb_rcvintpipe()创建。

对于批量 urb, 使用 usb_fill_bulk_urb()函数来初始化 urb, 如下所示:

```
void usb_fill_bulk_urb(struct urb *urb, struct usb_device *dev,
    unsigned int pipe, void *transfer_buffer,
    int buffer_length, usb_complete_t complete,
    void *context);
```

除了没有对应于调度间隔的 interval 参数以外, 该函数的参数和 usb_fill_int_urb()函数的参数含义相同。

上述函数参数中的 pipe 使用 usb_sndbulkpipe()或者 usb_rcvbulkpipe()函数来创建。

对于控制 urb, 使用 usb_fill_control_urb()函数来初始化 urb, 如下所示:

```
void usb_fill_control_urb(struct urb *urb, struct usb_device *dev,
    unsigned int pipe, unsigned char *setup_packet,
    void *transfer_buffer, int buffer_length,
    usb_complete_t complete, void *context);
```

除了增加了新的 setup_packet 参数以外, 该函数的参数和 usb_fill_bulk_urb()函数的参数含义相同。setup_packet 参数指向即将被发送到端点的设置数据包。

上述函数参数中的 pipe 使用 usb_sndctrlpipe()或 usb_rcvctrlpipe()函数来创建。

等时 urb 没有像中断、控制和批量 urb 的初始化函数, 我们只能手动地初始化 urb, 而后才能提交给 USB 核心。代码清单 20.14 给出了初始化等时 urb 的例子, 它来自 drivers/usb/media/usbvideo.c 文件。

代码清单 20.14 初始化等时 urb

```
1 for (i = 0; i < USBVIDEO_NUMSBUF; i++) {
2     int j, k;
3     struct urb *urb = uvd->sbuf[i].urb;
4     urb->dev = dev;
5     urb->context = uvd;
6     urb->pipe = usb_rcvisocpipe(dev, uvd->video_endp); /*端口*/
7     urb->interval = 1;
8     urb->transfer_flags = URB_ISO_ASAP; /*urb 被调度*/
9     urb->transfer_buffer = uvd->sbuf[i].data; /*传输 buffer*/
10    urb->complete = usbvideo_IsocIrq; /* 完成函数 */
11    urb->number_of_packets = FRAMES_PER_DESC; /*urb 中的等时传输数量*/
```

```

12 urb->transfer_buffer_length = uvd->iso_packet_len *FRAMES_PER_DESC;
13 for (j = k = 0; j < FRAMES_PER_DESC; j++, k += uvd->iso_packet_len) {
14     urb->iso_frame_desc[j].offset = k;
15     urb->iso_frame_desc[j].length = uvd->iso_packet_len;
16 }
17 }

```

(3) 被 USB 设备驱动提交给 USB 核心。

在完成第(1)、(2)步的创建和初始化 urb 后, urb 便可以提交给 USB 核心, 通过 `usb_submit_urb()` 函数来完成, 如下所示:

```
int usb_submit_urb(struct urb *urb, int mem_flags);
```

urb 参数是指向 urb 的指针, mem_flags 参数与传递给 `kmalloc()` 函数参数的意义相同, 它用于告知 USB 核心如何在此时分配内存缓冲区。

在提交 urb 到 USB 核心后, 直到完成函数被调用之前, 不要访问 urb 中的任何成员。

`usb_submit_urb()` 在原子上下文和进程上下文中都可以被调用, mem_flags 变量需根据调用环境进行相应的设置, 如下所示。

- **GFP_ATOMIC**: 在中断处理函数、底半部、tasklet、定时器处理函数以及 urb 完成函数中, 在调用者持有自旋锁或者读写锁时以及当驱动将 `current->state` 修改为非 `TASK_RUNNING` 时, 应使用此标志。
- **GFP_NOIO**: 在存储设备的块 I/O 和错误处理路径中, 应使用此标志;
- **GFP_KERNEL**: 如果没有任何理由使用 `GFP_ATOMIC` 和 `GFP_NOIO`, 就使用 `GFP_KERNEL`。

如果 `usb_submit_urb()` 调用成功, 即 urb 的控制权被移交给 USB 核心, 该函数返回 0; 否则, 返回错误号。

(4) 提交由 USB 核心指定的 USB 主机控制器驱动。

(5) 被 USB 主机控制器处理, 进行一次到 USB 设备的传送。

第(4)~(5)步由 USB 核心和主机控制器完成, 不受 USB 设备驱动的控制。

(6) 当 urb 完成, USB 主机控制器驱动通知 USB 设备驱动。

在如下 3 种情况下, urb 将结束, urb 完成函数将被调用。

- urb 被成功发送给设备, 并且设备返回正确的确认。如果 `urb->status` 为 0, 意味着对于一个输出 urb, 数据被成功发送; 对于一个输入 urb, 请求的数据被成功收到。
- 如果发送数据到设备或从设备接收数据时发生了错误, `urb->status` 将记录错误值。
- urb 被从 USB 核心“去除连接”, 这发生在驱动通过 `usb_unlink_urb()` 或 `usb_kill_urb()` 函数取消 urb, 或 urb 虽已提交, 而 USB 设备被拔出的情况下。

`usb_unlink_urb()` 和 `usb_kill_urb()` 这两个函数用于取消已提交的 urb, 其参数为要被取消的 urb 指针。对 `usb_unlink_urb()` 而言, 如果 urb 结构体中的 `URB_ASYNC_UNLINK` (即异步 unlink) 的标志被置位, 则对该 urb 的 `usb_unlink_urb()` 调用将立即返回, 具体的 unlink 动作将在后台进行。否则, 此函数一直等到 urb 被解开链接或结束时才返回。`usb_kill_urb()` 会彻底终止 urb 的生命周期, 它通常在设备的 `disconnect()` 函数中被调用。

当 urb 生命结束时 (处理完成或被解除链接), 通过 urb 结构体的 status 成员可以获知其原因, 如 0 表示传输成功, `-ENOENT` 表示被 `usb_kill_urb()` 杀死, `-ECONNRESET` 表示被 `usb_unlink_urb()`



杀死, `-EPROTO` 表示传输中发生了 `bitstuff` 错误或者硬件未能及时收到响应数据包, `-ENODEV` 表示 USB 设备已被移除, `-EXDEV` 表示等时传输仅完成了一部分等。

对以上 `urb` 的处理步骤进行一个总结, 图 20.5 给出了一个 `urb` 的整个处理流程, 虚线框的 `usb_unlink_urb()` 和 `usb_kill_urb()` 并非一定会发生, 它只是在 `urb` 正在被 USB 核心和主机控制器处理时, 被驱动程序取消的情况下才发生。

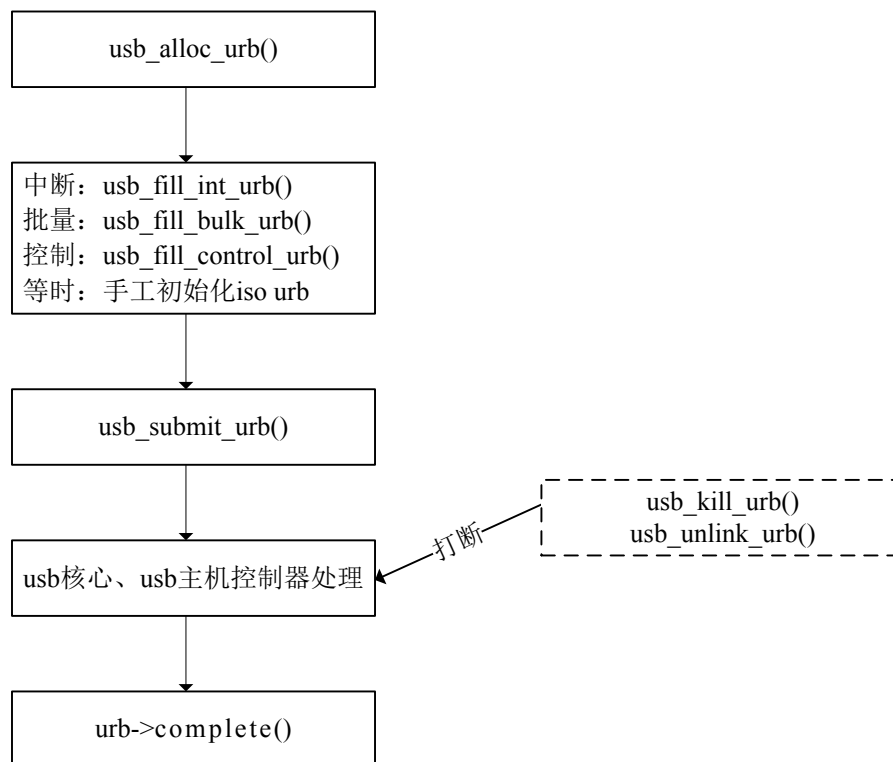


图 20.5 urb 处理流程

3. 简单的批量与控制 URB

有时 USB 驱动程序只是从 USB 设备上接收或向 USB 设备发送一些简单的数据, 这时候, 没有必要将 `urb` 创建、初始化、提交、完成处理的整个流程走一遍, 而可以使用两个更简单的函数, 如下所示。

(1) `usb_bulk_msg()`。

`usb_bulk_msg()` 函数创建一个 USB 批量 `urb` 并将它发送到特定设备, 这个函数是同步的, 它一直等待 `urb` 完成后才返回。`usb_bulk_msg()` 函数的原型为:

```
int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe,
                 void *data, int len, int *actual_length,
                 int timeout);
```

`usb_dev` 参数为批量消息要发送的 USB 设备的指针, `pipe` 为批量消息要发送到的 USB 设备的端点, `data` 参数为指向要发送或接收的数据缓冲区的指针, `len` 参数为 `data` 参数所指向的缓冲区的长度, `actual_length` 用于返回实际发送或接收的字节数, `timeout` 是发送超时, 以 `jiffies` 为单位, 0 意味着永远等待。

如果函数调用成功，返回 0；否则，返回 1 个负的错误值。

(2) `usb_control_msg()` 函数。

`usb_control_msg()` 函数与 `usb_bulk_msg()` 函数类似，不过它提供驱动发送和结束 USB 控制信息而非批量信息的能力，该函数的原型为：

```
int usb_control_msg(struct usb_device *dev, unsigned int pipe, __u8 request,
__u8 requesttype, __u16 value, __u16 index, void *data, __u16 size, int timeout);
```

`dev` 指向控制消息发往的 USB 设备，`pipe` 是控制消息要发往的 USB 设备的端点，`request` 是这个控制消息的 USB 请求值，`requesttype` 是这个控制消息的 USB 请求类型，`value` 是这个控制消息的 USB 消息值，`index` 是这个控制消息的 USB 消息索引值，`data` 指向要发送或接收的数据缓冲区，`size` 是 `data` 参数所指向的缓冲区的大小，`timeout` 是发送超时，以 jiffies 为单位，0 意味着永远等待。

参数 `request`、`requesttype`、`value` 和 `index` 与 USB 规范中定义的 USB 控制消息直接对应。

如果函数调用成功，该函数返回发送到设备或从设备接收到的字节数；否则，返回一个负的错误值。

对 `usb_bulk_msg()` 和 `usb_control_msg()` 函数的使用要特别慎重，由于它们是同步的，因此不能在中断上下文和持有自旋锁的情况下使用。而且，该函数也不能被任何其他函数取消，因此，务必要使得驱动程序的 `disconnect()` 函数掌握足够的信息，以判断和等待该调用的结束。

20.3.3 探测和断开函数

在 USB 设备驱动 `usb_driver` 结构体的探测函数中，应该完成如下工作。

- 探测设备的端点地址、缓冲区大小，初始化任何可能用于控制 USB 设备的数据结构。
- 把已初始化数据结构的指针保存到接口设备中。

`usb_set_intfdata()` 函数可以设置 `usb_interface` 的私有数据，这个函数的原型为：

```
void usb_set_intfdata (struct usb_interface *intf, void *data);
```

这个函数的“反函数”用于得到 `usb_interface` 的私有数据，其原型为：

```
void *usb_get_intfdata (struct usb_interface *intf);
```

- 注册 USB 设备。

如果是简单的字符设备，调用 `usb_register_dev()`，这个函数的原型为：

```
int usb_register_dev(struct usb_interface *intf,
struct usb_class_driver *class_driver);
```

上述函数中第二个参数为 `usb_class_driver` 结构体，这个结构体的定义如代码清单 20.15 所示。

代码清单 20.15 `usb_class_driver` 结构体

```
1 struct usb_class_driver {
2   char *name; /*sysfs 中用来描述设备名*/
3   struct file_operations *fops; /*文件操作结构体指针*/
4   int minor_base; /*开始次设备号*/
5 };
```

对于字符设备而言，`usb_class_driver` 结构体的 `fops` 成员中的 `write()`、`read()`、`ioctl()` 等函数的地位完全等同于本书第 6 章中的 `file_operations` 成员函数。

如果是其他类型的设备，如 tty 设备，则调用对应设备的注册函数。

在 USB 设备驱动 `usb_driver` 结构体的探测函数中，应该完成如下工作。



- 释放所有为设备分配的资源。
- 设置接口设备的数据指针为 NULL。
- 注销 USB 设备。

对于字符设备，可以直接调用 `usb_register_dev()` 函数的“反函数”，如下所示：

```
void usb_deregister_dev(struct usb_interface *intf,  
                        struct usb_class_driver *class_driver);
```

对于其他类型的设备，如 tty 设备，则调用对应设备的注销函数。

20.3.4 USB 骨架程序

Linux 内核源代码中的 `driver/usb/usb-skeleton.c` 文件为我们提供了一个最基础的 USB 驱动程序，即 USB 骨架程序，可被看做一个最简单的 USB 设备驱动实例。尽管具体 USB 设备驱动千差万别，但其骨架则万变不离其宗。

首先看看 USB 骨架程序的 `usb_driver` 结构体定义，如代码清单 20.16 所示。

代码清单 20.16 USB 骨架程序的 `usb_driver` 结构体

```
1 static struct usb_driver skel_driver = {  
2     .name = "skeleton",  
3     .probe = skel_probe,  
4     .disconnect = skel_disconnect,  
5     .suspend = skel_suspend,  
6     .resume = skel_resume,  
7     .pre_reset = skel_pre_reset,  
8     .post_reset = skel_post_reset,  
9     .id_table = skel_table,  
10    .supports_autosuspend = 1,  
11};
```

从上述代码第 9 行可以看出，它所支持的 USB 设备的列表数组为 `skel_table[]`，其定义如代码清单 20.17 所示。

代码清单 20.17 USB 骨架程序的 `id_table`

```
1 static struct usb_device_id skel_table [] = {  
2     { USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) },  
3     { } /* Terminating entry */  
4 };  
5 MODULE_DEVICE_TABLE(usb, skel_table);
```

对上述 `usb_driver` 的注册和注销发生在 USB 骨架程序的模块加载与卸载函数内，如代码清单 20.18 所示，其分别调用了 `usb_register()` 和 `usb_deregister()`。

代码清单 20.18 USB 骨架程序的模块加载与卸载函数

```
1 static int __init usb_skel_init(void)  
2 {  
3     int result;  
4  
5     /* 注册 USB 驱动 */  
6     result = usb_register(&skel_driver);  
7     if (result)  
8         err("usb_register failed. Error number %d", result);
```

```

9
10 return result;
11 }
12 static void __exit usb_skel_exit(void)
13 {
14     /* 注销 USB 驱动 */
15     usb_deregister(&skel_driver);
16 }

```

usb_driver 的 probe() 成员函数中, 会根据 usb_interface 的成员寻找第一个批量输入和输出端点, 将端点地址、缓冲区等信息存入为 USB 骨架程序定义的 usb_skel 结构体, 并将 usb_skel 实例的指针传入 usb_set_intfdata() 作为 USB 接口的私有数据, 最后, 它会注册 USB 设备, 如代码清单 20.19 所示。

代码清单 20.19 USB 骨架程序的探测函数

```

1 static int skel_probe(struct usb_interface *interface, const struct usb_device_id *id)
2 {
3     struct usb_skel *dev = NULL;
4     struct usb_host_interface *iface_desc;
5     struct usb_endpoint_descriptor *endpoint;
6     size_t buffer_size;
7     int i;
8     int retval = -ENOMEM;
9
10    /* 分配设备状态的内存并初始化 */
11    dev = kzalloc(sizeof(*dev), GFP_KERNEL);
12    if (dev == NULL) {
13        err("Out of memory");
14        goto error;
15    }
16    kref_init(&dev->kref);
17    sema_init(&dev->limit_sem, WRITES_IN_FLIGHT);
18
19    dev->udev = usb_get_dev(interface_to_usbdev(interface));
20    dev->interface = interface;
21
22    /* 设置端点信息 */
23    /* 仅使用第一个 bulk-in 和 bulk-out */
24    iface_desc = interface->cur_altsetting;
25    for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
26        endpoint = &iface_desc->endpoint[i].desc;
27
28        if (!dev->bulk_in_endpointAddr &&
29            ((endpoint->bEndpointAddress & USB_ENDPOINT_DIR_MASK)
30             == USB_DIR_IN) &&
31            ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
32             == USB_ENDPOINT_XFER_BULK)) {
33            /* 找到了一个批量 IN 端点 */
34            buffer_size = le16_to_cpu(endpoint->wMaxPacketSize);
35            dev->bulk_in_size = buffer_size;
36            dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
37            dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);
38            if (!dev->bulk_in_buffer) {

```



```
39         err("Could not allocate bulk_in_buffer");
40         goto error;
41     }
42 }
43
44 if (!dev->bulk_out_endpointAddr &&
45     ((endpoint->bEndpointAddress & USB_ENDPOINT_DIR_MASK)
46      == USB_DIR_OUT) &&
47     ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
48      == USB_ENDPOINT_XFER_BULK)) {
49     /* 找到了一个批量 OUT 端点 */
50     dev->bulk_out_endpointAddr = endpoint->bEndpointAddress;
51 }
52 }
53 if (!(dev->bulk_in_endpointAddr && dev->bulk_out_endpointAddr)) {
54     err("Could not find both bulk-in and bulk-out endpoints");
55     goto error;
56 }
57
58 /* 在设备结构中保存数据指针 */
59 usb_set_intfdata(interface, dev);
60
61 /* 注册 USB 设备 */
62 retval = usb_register_dev(interface, &skel_class);
63 if (retval) {
64     /* something prevented us from registering this driver */
65     err("Not able to get a minor for this device.");
66     usb_set_intfdata(interface, NULL);
67     goto error;
68 }
69
70 ...
71 }
```

usb_skel 结构体可以被看作一个私有数据结构体, 其定义如代码清单 20.20 所示, 应该根据具体的设备量身定制。

代码清单 20.20 USB 骨架程序的自定义数据结构 usb_skel

```
1 struct usb_skel {
2     struct usb_device *   udev;                /* 该设备的 usb_device 指针 */
3     struct usb_interface *interface;          /* 该设备的 usb_interface 指针 */
4     struct semaphore limit_sem;                /* 限制进程写的数量 */
5     unsigned char *       bulk_in_buffer;      /* 接收数据的缓冲区 */
6     size_t                 bulk_in_size;       /* 接收缓冲区大小 */
7     __u8                   bulk_in_endpointAddr; /* 批量 IN 端点的地址 */
8     __u8                   bulk_out_endpointAddr; /* 批量 OUT 端点的地址 */
9     ...
10    struct mutex           io_mutex;
11 };
```

USB 骨架程序的断开函数会完成探测函数相反的工作, 即设置接口数据为 NULL, 注销 USB 设备, 如代码清单 20.21 所示。

代码清单 20.21 USB 骨架程序的断开函数

```

1 static void skel_disconnect(struct usb_interface *interface)
2 {
3     struct usb_skel *dev;
4     int minor = interface->minor;
5
6     /* 阻止 skel_open() 与 skel_disconnect() 的竞争 */
7     lock_kernel();
8
9     dev = usb_get_intfdata(interface);
10    usb_set_intfdata(interface, NULL);
11
12    /* 注销 usb 设备, 释放次设备号 */
13    usb_deregister_dev(interface, &skel_class);
14
15    unlock_kernel();
16
17    /* 减少引用计数 */
18    kref_put(&dev->kref, skel_delete);
19
20    info("USB Skeleton #%d now disconnected", minor);
21 }

```

代码清单 20.18 第 62 行的 `usb_register_dev(interface, &skel_class)` 中第二个参数包含了字符设备的 `file_operations` 结构体指针, 而这个结构体中的成员实现也是 USB 字符设备的另一个组成成分。代码清单 20.22 给出了 USB 骨架程序的字符设备文件操作 `file_operations` 结构体的定义。

代码清单 20.22 USB 骨架程序的字符设备文件操作结构体

```

1 static const struct file_operations skel_fops = {
2     .owner = THIS_MODULE,
3     .read = skel_read,
4     .write = skel_write,
5     .open = skel_open,
6     .release = skel_release,
7     .flush = skel_flush,
8 };

```

由于只是一个象征性的骨架程序, `open()` 成员函数的实现非常简单, 它根据 `usb_driver` 和次设备号通过 `usb_find_interface()` 获得 USB 接口, 之后通过 `usb_get_intfdata()` 获得接口的私有数据并赋予 `file->private_data`, 如代码清单 20.23 所示。

代码清单 20.23 USB 骨架程序的字符设备打开函数

```

1 static int skel_open(struct inode *inode, struct file *file)
2 {
3     struct usb_skel *dev;
4     struct usb_interface *interface;
5     int subminor;
6     int retval = 0;
7
8     subminor = iminor(inode);
9
10    interface = usb_find_interface(&skel_driver, subminor); /* 获得接口数据 */
11    ...

```



```
12
13 dev = usb_get_intfdata(interface);
14 ...
15
16 kref_get(&dev->kref);
17
18 mutex_lock(&dev->io_mutex);
19
20 if (!dev->open_count++) {
21     retval = usb_autopm_get_interface(interface);
22     if (retval) {
23         dev->open_count--;
24         mutex_unlock(&dev->io_mutex);
25         kref_put(&dev->kref, skel_delete);
26         goto exit;
27     }
28 }
29
30 file->private_data = dev; /* 将接口数据保存在 file->private_data 中 */
31 mutex_unlock(&dev->io_mutex);
32 ...
33 }
```

由于在 `open()` 函数中并没有申请任何软件和硬件资源, 因此与 `open()` 函数对应的 `release()` 函数不用进行资源的释放, 它进行减少在 `open()` 中增加的引用计数等工作。

接下来要分析的是读写函数, 前面已经提到, 在访问 USB 设备的时候, 贯穿于其中的“中枢神经”是 `urb` 结构体。

在 `skel_write()` 函数中进行的关于 `urb` 的操作与 20.3.2 小节的描述完全对应, 即进行了 `urb` 的分配 (调用 `usb_alloc_urb()`)、初始化 (调用 `usb_fill_bulk_urb()`) 和提交 (调用 `usb_submit_urb()`) 的操作, 如代码清单 20.24 所示。

代码清单 20.24 USB 骨架程序的字符设备写函数

```
1 static ssize_t skel_write(struct file *file, const char *user_buffer, size_t count, loff_t *ppos)
2 {
3     ...
4     /* 创建 urb、urb 的缓冲区, 将数据复制给 urb */
5     urb = usb_alloc_urb(0, GFP_KERNEL);
6     ...
7
8     buf = usb_buffer_alloc(dev->udev, writesize, GFP_KERNEL, &urb->transfer_dma);
9     ...
10
11     if (copy_from_user(buf, user_buffer, writesize)) {
12         retval = -EFAULT;
13         goto error;
14     }
15     ...
16
17     /* 初始化 urb */
18     usb_fill_bulk_urb(urb, dev->udev,
19         usb_sndbulkpipe(dev->udev, dev->bulk_out_endpointAddr),
```

```

20         buf, writesize, skel_write_bulk_callback, dev);
21     urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
22     usb_anchor_urb(urb, &dev->submitted);
23
24     /* 将数据发送到批量端口 */
25     retval = usb_submit_urb(urb, GFP_KERNEL);
26     ...
27
28     /* 释放对 urb 的引用, USB 将最终完全释放之 */
29     usb_free_urb(urb);
30
31     return writesize;
32
33     ...
34 }

```

写函数中发起的 urb 结束后, 其完成函数 `skel_write_bulk_callback()` 将被调用, 它会进行 `urb->status` 的判断, 如代码清单 20.25 所示。

代码清单 20.25 USB 骨架程序的字符设备写操作完成函数

```

1 static void skel_write_bulk_callback(struct urb *urb, struct pt_regs *regs)
2 {
3     struct usb_skel *dev;
4
5     dev = (struct usb_skel *)urb->context;
6
7     ...
8
9     /* 释放被分配的内存 */
10    usb_buffer_free(urb->dev, urb->transfer_buffer_length,
11                    urb->transfer_buffer, urb->transfer_dma);
12    up(&dev->limit_sem);
13 }

```

USB 骨架程序的字符设备读函数并没有进行类似写函数的一系列针对 urb 的操作, 而是简单地调用 `usb_bulk_msg()` 发起一次同步 urb 传输操作, 如代码清单 20.26 所示。

代码清单 20.26 USB 骨架程序的字符设备读函数

```

1 static ssize_t skel_read(struct file *file, char *buffer, size_t count, loff_t *ppos)
2 {
3     struct usb_skel *dev;
4     int retval = 0;
5     int bytes_read;
6
7     dev = (struct usb_skel *)file->private_data;
8
9     /* 从设备进行一次阻塞的批量读 */
10    retval = usb_bulk_msg(dev->udev,
11                           usb_rcvbulkpipe(dev->udev, dev->bulk_in_endpointAddr),
12                           dev->bulk_in_buffer,
13                           min(dev->bulk_in_size, count),
14                           &bytes_read, 10000);

```



```
15
16  /* 如果读成功, 将数据复制到用户空间 */
17  if (!retval) {
18      if (copy_to_user(buffer, dev-bulk_in_buffer, bytes_read))
19          retval = -EFAULT;
20      else
21          retval = bytes_read;
22  }
23
24  return retval;
25 }
```

20.3.5 实例：USB 键盘驱动

在 Linux 系统中, 键盘被认定为标准输入设备, 对于一个 USB 键盘而言, 其驱动主要由两部分组成: `usb_driver` 的成员函数以及输入设备驱动的 `input_event` 获取和报告。

在 USB 键盘设备驱动的模块加载和卸载函数中, 将分别注册和注销对应于 USB 键盘的 `usb_driver` 结构体 `usb_kbd_driver`, 代码清单 20.27 所示为模块加载与卸载函数以及 `usb_kbd_driver` 结构体的定义。

代码清单 20.27 USB 键盘设备驱动的模块加载与卸载函数及 `usb_driver` 结构体

```
1 static int __init usb_kbd_init(void)
2 {
3     int result = usb_register(&usb_kbd_driver);
4     ...
5 }
6
7 static void __exit usb_kbd_exit(void)
8 {
9     usb_deregister(&usb_kbd_driver);
10 }
11
12 static struct usb_driver usb_kbd_driver =
13 {
14     .name = "usbkbd",
15     .probe = usb_kbd_probe,
16     .disconnect = usb_kbd_disconnect,
17     .id_table = usb_kbd_id_table,
18 };
19
20 static struct usb_device_id usb_kbd_id_table [] = {
21     { USB_INTERFACE_INFO(USB_INTERFACE_CLASS_HID, USB_INTERFACE_SUBCLASS_BOOT,
22         USB_INTERFACE_PROTOCOL_KEYBOARD) },
23     { }
24 };
25 MODULE_DEVICE_TABLE (usb, usb_kbd_id_table);
```

在 `usb_driver` 的探测函数中, 将进行 `input` 设备的初始化和注册, USB 键盘要使用的中断 `urb` 和控制 `urb` 的初始化, 并设置接口的私有数据, 如代码清单 20.28 所示。

代码清单 20.28 USB 键盘设备驱动的探测函数

```

1 static int usb_kbd_probe(struct usb_interface *iface, const struct
2   usb_device_id *id)
3 {
4   ...
5   pipe = usb_rcvintpipe(dev, endpoint->bEndpointAddress);
6   maxp = usb_maxpacket(dev, pipe, usb_pipeout(pipe));
7
8   kbd = kzalloc(sizeof(struct usb_kbd), GFP_KERNEL);
9   input_dev = input_allocate_device(); /* 分配 input_dev 结构体 */
10
11   ...
12   /* 输入设备初始化 */
13   input_dev->name = kbd->name;
14   input_dev->phys = kbd->phys;
15   usb_to_input_id(dev, &input_dev->id);
16   input_dev->cdev.dev = &iface->dev;
17   input_dev->private = kbd;
18
19   input_dev->evbit[0] = BIT(EV_KEY) | BIT(EV_LED) | BIT(EV_REP);
20   input_dev->ledbit[0] = BIT(LED_NUML) | BIT(LED_CAPSL) |
21     BIT(LED_SCROLLL) | BIT(LED_COMPOSE) | BIT(LED_KANA);
22
23   ...
24   /* 初始化中断 urb */
25   usb_fill_int_urb(kbd->irq, dev, pipe, kbd->new, (maxp > 8 ? 8 : maxp),
26     usb_kbd_irq, kbd, endpoint->bInterval);
27   kbd->irq->transfer_dma = kbd->new_dma;
28   kbd->irq->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
29
30   ...
31   /* 初始化控制 urb */
32   usb_fill_control_urb(kbd->led, dev, usb_sndctrlpipe(dev, 0), (void*) kbd->cr,
33     kbd->leds, 1, usb_kbd_led, kbd);
34   ...
35   input_register_device(kbd->dev); /* 注册输入设备 */
36
37   usb_set_intfdata(iface, kbd); /* 设置接口私有数据 */
38   return 0;
39   ...
40 }

```

在 `usb_driver` 的断开函数中，将设置接口私有数据为 `NULL`、终止已提交的 `urb` 并注销输入设备，如代码清单 20.29 所示。

代码清单 20.29 USB 键盘设备驱动的断开函数

```

1 static void usb_kbd_disconnect(struct usb_interface *intf)
2 {
3   struct usb_kbd *kbd = usb_get_intfdata(intf);
4
5   usb_set_intfdata(intf, NULL); /* 设置接口私有数据为 NULL */
6   if (kbd)
7   {
8     usb_kill_urb(kbd->irq); /* 终止 urb */

```



```
9     input_unregister_device(kbd-dev); /* 注销输入设备 */
10     usb_kbd_free_mem(interface_to_usbdev(intf), kbd);
11     kfree(kbd);
12 }
13 }
```

键盘主要依赖于中断传输模式，在键盘中断 urb 的完成函数 `usb_kbd_irq()` 中（通过代码清单 20.28 的第 26 行可以看出），将会通过 `input_report_key()` 报告按键事件，通过 `input_sync()` 报告同步事件，如代码清单 20.30 所示。

代码清单 20.30 USB 键盘设备驱动的中断 urb 完成函数

```
1 static void usb_kbd_irq(struct urb *urb, struct pt_regs *regs)
2 {
3     struct usb_kbd *kbd = urb->context;
4     int i;
5
6     switch (urb->status) {
7     case 0: /* 成功 */
8         break;
9     case - ECONNRESET: /* unlink */
10    case - ENOENT:
11    case - ESHUTDOWN:
12        return ;
13    default: /* 错误 */
14        goto resubmit;
15    }
16    /* 获得键盘扫描码并报告按键事件，这里没有列出细节 */
17
18    input_report_key(kbd->dev, usb_kbd_keycode[kbd->old[i]], 0);
19    ...
20    input_report_key(kbd->dev, usb_kbd_keycode[kbd->new[i]], 1);
21    ...
22    input_sync(kbd->dev); /* 报告同步事件 */
23
24    resubmit:
25        i = usb_submit_urb (urb, GFP_ATOMIC);
26        ...
27 }
```

从 USB 键盘驱动的例子中，我们进一步看到了 `usb_driver` 本身只是起一个挂接总线的作用，而具体的设备类型的驱动仍然是工作的主体，例如键盘就是 `input`、USB 串口就是 `tty`，只是在这些设备底层进行硬件访问的时候，调用的都是 URB 相关的接口，URB 这套 USB 核心层 API 的存在，使我们无需关心底层 USB 主机控制器的具体细节，因此，USB 设备驱动也变得与平台无关，同样的驱动可应用于不同的 SoC。

20.4 USB UDC 与 gadget 驱动

20.4.1 UDC 和 gadget 驱动关键数据结构与 API

这里的 USB 设备控制器 (UDC) 驱动指作为其他 USB 主机控制器外设的 USB 硬件设备上底

层硬件控制器的驱动,该硬件和驱动负责将一个 USB 设备依附于一个 USB 主机控制器上。例如,当某运行 Linux 的手机作为 PC 的 U 盘时,手机中的底层 USB 控制器行使 USB 设备控制器的功能,这时候运行在底层的是 UDC 驱动,而手机要成为 U 盘,在 UDC 驱动之上仍然需要另外一个驱动,对于 USB 大容量存储器为 file storage 驱动,这一驱动称为 gadget 驱动。从图 20.1 左边可以看出,USB 设备驱动调用 USB 核心的 API,因此具体驱动与 SoC 无关;同样,从图 20.1 右边可以看出,USB gadget 驱动调用通用的 gadget API,因此具体 gadget 驱动也变得与 SoC 无关。软件分层设计的好处再一次得到了深刻的体现。

UDC 驱动和 gadget 驱动都位于内核的 drivers/usb/gadget 目录,omap_udc.c、pxa27x_udc.c、m66592-udc.c、s3c2410_udc.c 等是对应 SoC 平台上的 UDC 驱动,ether.c、f_serial.c、file_storage.c 等文件实现了一些 gadget 驱动,重要的 gadget 驱动如下所示。

Gadget Zero: 该驱动用于测试 udc 驱动,它会帮助您通过 USB-IF 测试。

Ethernet over USB: 该驱动模拟以太网网口,它支持多种运行方式——CDC Ethernet (实现标准的 Communications Device Class "Ethernet Model" 协议)、CDC Subset (由于硬件受限,仅实现 CDC Ethernet 的一个子集,不含 altsetting) 以及 RNDIS (微软公司对 CDC Ethernet 的变种实现) 这几种模式。

File-backed Storage Gadget: 最常见的 U 盘功能实现。

Serial Gadget: 包括 Generic Serial 实现 (只需要 Bulk-in/Bulk-out 端点+ep0) 和 CDC ACM 规范实现。内核源代码中的 Documentation/usb/gadget_serial.txt 文档讲解了如何将 Serial Gadget 与 Windows 和 Linux 主机连接。

Gadget MIDI: 暴露 ALSA MIDI 接口。

另外,drivers/usb/gadget 源代码还实现了一个 Gadget 文件系统(GadgetFS),可以将 Gadget API 接口暴露给应用层,以便在应用层实现用户空间的驱动。

在 USB 设备控制器与 gadget 驱动中,我们主要关心几个核心的数据结构,这些数据结构包括描述一个 USB 设备控制器的 usb_gadget、描述一个 gadget 驱动的 usb_gadget_driver、表示一个传输请求的 usb_request (与从主机端看到的 urb 相似)、描述一个端点的 usb_ep、描述端点操作的 usb_ep_ops 结构体。UDC 和 gadget 驱动围绕这些数据结构及其成员函数而展开,代码清单 20.31 列出了这些关键的数据结构,都定义于 include/linux/usb/gadget.h 文件。

代码清单 20.31 UDC 和 gadget 驱动关键数据结构

```

1 struct usb_gadget {
2     /* 针对 gadget 驱动只读 */
3     const struct usb_gadget_ops *ops; /* 访问硬件的函数 */
4     struct usb_ep *ep0; /* endpoint 0, setup 使用 */
5     struct list_head ep_list; /* 其他 endpoint 的列表 */
6     enum usb_device_speed speed;
7     unsigned is_dualspeed:1;
8     unsigned is_otg:1;
9     unsigned is_a_peripheral:1;
10    unsigned b_hnp_enable:1; /* A-HOST 使能了 HNP 支持 */
11    unsigned a_hnp_support:1; /* A-HOST 支持 HNP */
12    unsigned a_alt_hnp_support:1;
13    const char *name;

```



```
14 struct device                dev;
15 };
16
17 struct usb_gadget_driver {
18     char                *function; /* 描述 gadget 功能的字符串 */
19     enum usb_device_speed    speed;
20     int                  (*bind)(struct usb_gadget *); /* 当驱动与 gadget 绑定时调用 */
21     void                 (*unbind)(struct usb_gadget *);
22     int                  (*setup)(struct usb_gadget *, /* 处理硬件驱动未处理的 ep0 请求 */
23                                const struct usb_ctrlrequest *);
24     void                 (*disconnect)(struct usb_gadget *);
25     void                 (*suspend)(struct usb_gadget *);
26     void                 (*resume)(struct usb_gadget *);
27
28     struct device_driver    driver;
29 };
30
31 struct usb_request {
32     void                *buf;
33     unsigned            length;
34     dma_addr_t          dma;
35
36     unsigned            no_interrupt:1;
37     unsigned            zero:1;
38     unsigned            short_not_ok:1;
39
40     void                (*complete)(struct usb_ep *ep,
41                                struct usb_request *req); /* 当请求完成时调用的函数 */
42     void                *context;
43     struct list_head    list;
44
45     int                 status;
46     unsigned            actual;
47 };
48
49 struct usb_ep {
50     void                *driver_data;
51
52     const char          *name;
53     const struct usb_ep_ops *ops;
54     struct list_head    ep_list;
55     unsigned            maxpacket:16;
56 };
57
58 struct usb_ep_ops {
59     int (*enable)(struct usb_ep *ep,
60                 const struct usb_endpoint_descriptor *desc);
61     int (*disable)(struct usb_ep *ep);
62
63     struct usb_request *(*alloc_request)(struct usb_ep *ep,
64                                         gfp_t gfp_flags);
65     void (*free_request)(struct usb_ep *ep, struct usb_request *req);
66 }
```

```

67  int (*queue) (struct usb_ep *ep, struct usb_request *req,
68              gfp_t gfp_flags); /* 将 usb_request 提交给 endPoint, 进行数据传输 */
69  int (*dequeue) (struct usb_ep *ep, struct usb_request *req);
70
71  int (*set_halt) (struct usb_ep *ep, int value);
72  int (*set_wedge) (struct usb_ep *ep);
73
74  int (*fifo_status) (struct usb_ep *ep);
75  void (*fifo_flush) (struct usb_ep *ep);
76 };

```

在具体的 UDC 驱动中, 需要封装 `usb_gadget` 和每个端点 `usb_ep`, 实现端点的 `usb_ep_ops`, 完成 `usb_request`。另外, `usb_gadget_register_driver()`、`usb_gadget_unregister_driver()` 这两个 API 需要由 UDC 驱动提供, `gadget` 驱动会调用它们, 其原型分别为:

```

int usb_gadget_register_driver(struct usb_gadget_driver *driver);
int usb_gadget_unregister_driver(struct usb_gadget_driver *driver);

```

`usb_gadget_register_driver()` 通常在 `gadget` 驱动模块初始化时调用, 该函数中通常会调用 `driver→bind()` 函数, 将该 `usb_gadget_driver` 与具体的 `gadget` 绑定, 该函数最好放在 `init` 节内。

`usb_gadget_unregister_driver()` 通常在 `gadget` 驱动模块卸载时调用, 告诉底层的 UDC 驱动不再投入工作。如果 UDC 正与 USB 主机连接, 会先调用 `driver→disconnect()` 函数, 在 `usb_gadget_unregister_driver()` 函数返回前, 也需调用 `driver→unbind()` 函数。所以 `unbind()` 函数适合放在 `exit` 节。

在 `linux/usb/gadget.h` 中, 还封装了一些常用的 API, 如下所示。

(1) 使能和禁止端点。

```

static inline int usb_ep_enable(struct usb_ep *ep,
                               const struct usb_endpoint_descriptor *desc);
static inline int usb_ep_disable(struct usb_ep *ep);

```

它们分别调用了 “`ep→ops→enable(ep, desc);`” 和 “`ep→ops→disable(ep);`”。

(2) 分配和释放 `usb_request`。

```

static inline struct usb_request *usb_ep_alloc_request(struct usb_ep *ep,
                                                      gfp_t gfp_flags);
static inline void usb_ep_free_request(struct usb_ep *ep,
                                       struct usb_request *req);

```

它们分别调用了 “`ep→ops→alloc_request(ep, gfp_flags);`” 和 “`ep→ops→free_request(ep, req);`”。用于分配和释放一个依附于某端点的 `usb_request`。

(3) 提交和取消 `usb_request`。

```

static inline int usb_ep_queue(struct usb_ep *ep,
                              struct usb_request *req, gfp_t gfp_flags);
static inline int usb_ep_dequeue(struct usb_ep *ep, struct usb_request *req);

```

它们分别调用 “`ep→ops→queue(ep, req, gfp_flags);`” 和 “`ep→ops→dequeue(ep, req);`”。`usb_ep_queue` 函数告诉 UDC 完成 `usb_request` (读写 `buffer`), 当请求被完成后, 该请求对应的 `completion` 函数会被调用。

(4) 端点 FIFO 管理。

```

static inline int usb_ep_fifo_status(struct usb_ep *ep);
static inline void usb_ep_fifo_flush(struct usb_ep *ep);

```

前者调用 “`ep→ops→fifo_status(ep)`” 返回目前 FIFO 中的字节数, 后者调用 “`ep→ops→fifo_flush(ep)`”



以 flush 掉 FIFO 中的数据。

(5) 返回目前的帧号。

```
static inline int usb_gadget_frame_number(struct usb_gadget *gadget);
```

它调用 “`gadget→ops→get_frame(gadget)`” 返回目前的帧号，正常为自 SOF 以来的一个 11 位数，如果设备不支持该能力，则返回出错码。

(6) 管理配置描述符。

```
int usb_descriptor_fillbuf(void *, unsigned,
                           const struct usb_descriptor_header **);
int usb_gadget_config_buf(const struct usb_config_descriptor *config,
                          void *buf, unsigned buflen, const struct usb_descriptor_header **desc);
static inline void usb_free_descriptors(struct usb_descriptor_header **v);
```

其他 API 还包括 `usb_gadget_vbus_connect()`、`usb_gadget_vbus_disconnect()`、`usb_ep_set_halt()`、`usb_ep_clear_halt()`、`usb_gadget_vbus_draw()`、`usb_gadget_wakeup()`、`usb_gadget_set_selfpowered()`、`usb_gadget_clear_selfpowered()`、`usb_gadget_connect()`、`usb_gadget_disconnect`、`usb_ep_set_wedge()` 等，处理 USB 总线上的一些电源管理、OTG 协议等，详见 `include/linux/usb/gadget.h`。

20.4.2 实例：S3C6410 USB 2.0 的 UDC 驱动

S3C6410 除了一个 USB1.1 主机接口以外，还包括一个 USB 2.0 支持 OTG 的控制器。它既支持主机，又支持外设功能。当使能 `USB_S3C_OTG_HOST` 选项（即 “S3C USB OTG Host support” 菜单时），`drivers/usb/host/s3c-otg` 目录中的源码被选中，实现主机控制器驱动；当使能 `USB_GADGET_S3C_OTGD`（即 “S3C HS USB OTG Device”）时，`drivers/usb/gadget/s3c_udc_otg.c` 被编译，成为一个 UDC。

`drivers/usb/gadget/s3c_udc.h` 中定义了一个 `s3c_udc` 结构体，将 S3C6410 UDC 的 `gadget`、`usb_gadget_driver`、`endpoint` 等封装在一起，而 `drivers/usb/gadget/s3c_udc_otg.c` 则定义了一个它的实例，如代码清单 20.32 所示。

代码清单 20.32 S3C6410 UDC 驱动的 `s3c_udc` 结构体以及实例

```
1 struct s3c_udc {
2     struct usb_gadget gadget;
3     struct usb_gadget_driver *driver;
4     struct platform_device *dev;
5     spinlock_t lock;
6
7     int ep0state;
8     struct s3c_ep ep[S3C_MAX_ENDPOINTS];
9
10    unsigned char usb_address;
11
12    unsigned req_pending:1, req_std:1, req_config:1;
13 };
14
15 static struct s3c_udc memory = {
16     .usb_address = 0,
17
18     .gadget = {
19         .ops = &s3c_udc_ops,
```

```

20     .ep0 = &memory.ep[0].ep,
21     .name = driver_name,
22     .dev = {
23         .bus_id = "gadget",
24         .release = nop_release,
25     },
26 },
27
28 /* 控制器 endpoint */
29 .ep[0] = {
30     .ep = {
31         .name = ep0name,
32         .ops = &s3c_ep_ops,
33         .maxpacket = EP0_FIFO_SIZE,
34     },
35     .dev = &memory,
36
37     .bEndpointAddress = 0,
38     .bmAttributes = 0,
39
40     .ep_type = ep_control,
41     .fifo = (unsigned int) S3C_UDC_OTG_EP0_FIFO,
42 },
43
44 /* endpoints */
45 .ep[1] = {
46     .ep = {
47         .name = "ep1-bulk",
48         .ops = &s3c_ep_ops,
49         .maxpacket = EP_FIFO_SIZE,
50     },
51     .dev = &memory,
52
53     .bEndpointAddress = USB_DIR_OUT | 1,
54     .bmAttributes = USB_ENDPOINT_XFER_BULK,
55
56     .ep_type = ep_bulk_out,
57     .fifo = (unsigned int) S3C_UDC_OTG_EP1_FIFO,
58 },
59
60 .ep[2] = {
61     ...
62     .bEndpointAddress = USB_DIR_IN | 2,
63     .bmAttributes = USB_ENDPOINT_XFER_BULK,
64
65     .ep_type = ep_bulk_in,
66 },
67
68 .ep[3] = {
69     ...
70     .bEndpointAddress = USB_DIR_IN | 3,
71     .bmAttributes = USB_ENDPOINT_XFER_INT,
72     ...
73 },
74 .ep[4] = {

```



```
75     ...
76     .bEndpointAddress = USB_DIR_OUT | 4,
77     .bmAttributes = USB_ENDPOINT_XFER_BULK,
78     ...
79     },
80     .ep[5] = {
81         ...
82         .bEndpointAddress = USB_DIR_IN | 5,
83         .bmAttributes = USB_ENDPOINT_XFER_BULK,
84
85         .ep_type = ep_bulk_in,
86     },
87     .ep[6] = {
88         ...
89         .bEndpointAddress = USB_DIR_IN | 6,
90         .bmAttributes = USB_ENDPOINT_XFER_INT,
91
92         .ep_type = ep_interrupt,
93     },
94     .ep[7] = {
95         ...
96         .bEndpointAddress = USB_DIR_OUT | 7,
97         .bmAttributes = USB_ENDPOINT_XFER_BULK,
98
99         .ep_type = ep_bulk_out,
100    },
101    ...
102 };
```

上述代码的第 18~26 行是对 UDC 整体的描述以及操作,从第 29 行起,描述的就是 S3C6410 UDC 的各个端点的类型、地址和操作。`s3c_udc` 类型的 `memory` 是一个全局变量,实际上,在 `drivers/usb/gadget/s3c_udc_otg.c` 是一个基于 `platform` 的设备驱动的 `probe()` 过程中,还会进一步初始化 `memory` 中的各成员,例如通过 `list_add_tail()` 将各 `endpoint` 添加到 `ep_list` 等。

`drivers/usb/gadget/s3c_udc_otg.c` 其他主要工作是根据 `endpoint` 的列表配置硬件和完成 `usb_request` 数据发送和接收请求,实现 `s3c_udc_ops` 和 `s3c_ep_ops` 这两个结构体的成员函数,实现 `usb_gadget_register_driver()`、`usb_gadget_unregister_driver()` 这两个 API 等。

20.4.3 实例：file storage gadget 驱动

`file storage gadget` 驱动由 `drivers/usb/gadget/file_storage.c` 文件实现,它完成的主要工作如下。

- (1) 实现 `usb_gadget_driver` 实例及其中的成员函数 `bind()`、`unbind()`、`disconnect()`、`setup()` 等。
- (2) 准备作为 U 盘外设的设备描述符 `usb_device_descriptor`、配置描述符 `usb_config_descriptor`、接口描述符 `usb_interface_descriptor`、端点描述符 `usb_endpoint_descriptor` 等。
- (3) 完成与虚拟文件系统 VFS 的交互,将文件作为 U 盘映像,透过 `vfs_read()`、`vfs_write()` 读写文件,并透过 `usb_request` 在主机与 `gadget` 间交换数据。

在 LDD6410 开发板上,USB 2.0 OTG 的原理如图 20.6 所示。当 LDD6410 开发板内核配置选中 `drivers/usb/gadget/s3c_udc_otg.c` 和 `drivers/usb/gadget/file_storage.c` 的情况下,通过加载 `g_file_storage` 模块并传入一个映像名作为参数,即可开启 LDD6410 开发板的 U 盘功能。

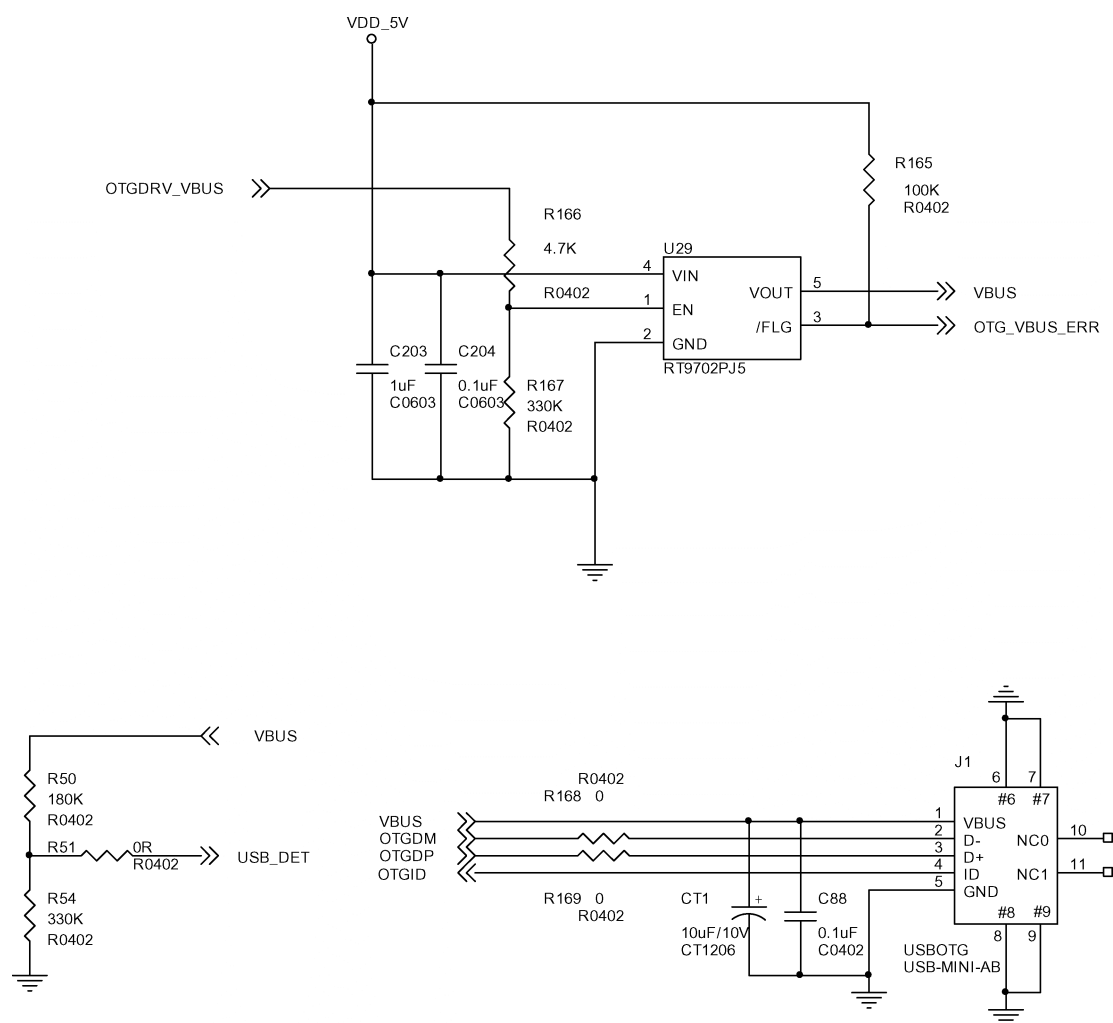


图 20.6 LDD6410 开发板 USB 2.0 OTG 接口原理图

LDD6410 开发板的/demo 目录包含一个做好的映像 vfat.img, 因此加载 g_file_storage 的命令为:

```
# modprobe g_file_storage file=/demo/vfat.img stall=0 removable=1
g_file_storage gadget: Filebacked Storage Gadget, version: 7 August 2007
g_file_storage gadget: Number of LUNs=1 g_file_storage gadgetlun0: ro=0, file:
/demo/vfat.img Registered gadget driver 'g_file_storage'
```

vfat.img 可以在 PC 上通过 dd 和 mkfs.vfat 命令得到:

```
$ dd if=/dev/zero of=vfat.img bs=1M count=20
20+0 records in
20+0 records out
20971520 bytes (21 MB) copied, 0.195482 s, 107 MB/s
$ sudo losetup /dev/loop0 vfat.img
$ sudo mkfs.vfat /dev/loop0
mkfs.vfat 2.11 (12 Mar 2005)
Loop device does not match a floppy size, using default hd params
$ mkdir vfat_mount_point
$ sudo mount -t vfat /dev/loop0 vfat_mount_point
```



这样之后可以把需要的文件拷入 `vfat_mount_point` 目录, 完成后 `umount vfat_mount_point` 目录并删除 `loop0`:

```
$ sudo umount vfat_mount_point
$ sudo losetup -d /dev/loop0
```

20.5 USB OTG 驱动

USB OTG 标准在完全兼容 USB 2.0 标准的基础上, 它允许设备既可作为主机, 也可作为外设操作, OTG 新增了主机通令协议 (HNP) 和对话请求协议 (SRP)。

在 OTG 中, 初始主机设备称为 A 设备, 外设称为 B 设备。可用电缆的连接方式来决定初始角色。两用设备使用新型 mini-AB 插座, 从而使 mini-A 插头、mini-B 插头和 mini-AB 插座增添了第 5 个引脚 (ID), 以用于识别不同的电缆端点。mini-A 插头中的 ID 引脚接地, mini-B 插头中的 ID 引脚浮空。当 OTG 设备检测到接地的 ID 引脚时, 表示默认的是 A 设备 (主机), 而检测到 ID 引脚浮空的设备则认为是 B 设备 (外设)。系统一旦连接后, OTG 的角色还可以更换, 采用新的 HNP 协议。而 SRP 允许 B 设备请求 A 设备打开 VBUS 电源并启动一次对话。一次 OTG 对话可通过 A 设备提供 VBUS 电源的时间来确定。

自从 Linux 2.6.9 开始, OTG 相关源代码已经被包含在内核中, 新增的主要内容包括:

(1) UDC 驱动端添加的 OTG 相关属性和函数。

```
struct usb_gadget {
    ...
    unsigned      is_otg:1;
    unsigned      is_a_peripheral:1;
    unsigned      b_hnp_enable:1;
    unsigned      a_hnp_support:1;
    unsigned      a_alt_hnp_support:1;
    ...
};

int usb_gadget_vbus_connect(struct usb_gadget *gadget);
int usb_gadget_vbus_disconnect(struct usb_gadget *gadget);

int usb_gadget_vbus_draw(struct usb_gadget *gadget, unsigned mA);

/* 控制 USB D+的 pullup */
int usb_gadget_connect(struct usb_gadget *gadget);
int usb_gadget_disconnect(struct usb_gadget *gadget);

int usb_gadget_wakeup(struct usb_gadget *gadget);
```

(2) gadget 驱动端添加的 OTG 相关属性和函数。

如果 `gadget→is_otg` 字段为真, 则增加一个 OTG 描述符; 通过 `printk()`、LED 等方式报告 HNP 可用; 当 `suspend` 开始时, 通过用户界面报告 HNP 切换开始 (B-Peripheral 到 B-Host 或 A-Peripheral to A-Host)。

(3) 主机侧添加的 OTG 相关属性和函数。

USB 核心中新增了关于 OTG 设备枚举的信息:

```

struct usb_bus {
    ...
    u8 otg_port;           /* 0, or index of OTG/HNP port */
    unsigned is_b_host:1;  /* true during some HNP roleswitches */
    unsigned b_hnp_enable:1; /* OTG: did A-Host enable HNP? */
    ...
};

```

为了实现 HNP 需要的 suspend/resume, 新增如下通用接口:

```

int usb_suspend_device(struct usb_device *dev, u32 state);
int usb_resume_device(struct usb_device *dev);

```

(4) 新增 OTG 控制器 otg_transceiver。

```

struct otg_transceiver {
    struct device      *dev;
    const char         *label;

    u8                  default_a;
    enum usb_otg_state  state;

    struct usb_bus      *host;
    struct usb_gadget *gadget;
    /* to pass extra port status to the root hub */
    ul6                  port_status;
    ul6                  port_change;
    /* bind/unbind 主机控制器 */
    int (*set_host)(struct otg_transceiver *otg,
                    struct usb_bus *host);
    /* bind/unbind 设备控制器 */
    int (*set_peripheral)(struct otg_transceiver *otg,
                          struct usb_gadget *gadget);
    /* 对 B 设备有效 */
    int (*set_power)(struct otg_transceiver *otg,
                    unsigned mA);
    /* 针对 B 设备: 与 A-Host 进入一个 session */
    int (*start_srp)(struct otg_transceiver *otg);
    /* 开始/继续 HNP 角色切换 */
    int (*start_hnp)(struct otg_transceiver *otg);
};

```

目前, 完整实现 OTG 支持的驱动非常少, 例如目前 S3C6410 的 USB 2.0 控制器驱动就没有完整实现 OTG, 因此, 我们无法在运行时动态切换 S3C6410 的身份, 而 TI OMAP 处理器的 OTG 支持比较完整。而真正支持完整 OTG 功能的产品也非常少, Nokia N810 internet Tablet 是其中之一, 它使用的 SoC 是 OMAP 2420。其他大多号称支持 OTG 的产品实际上并未完整实现 HNP 和 SRP。

20.6 总结

USB 驱动分为 USB 主机驱动和 USB 设备驱动, 如果系统的 USB 主机控制器符合 OHCI 等标准, 这主机驱动的绝大部分工作都可以沿用通用的代码。



对于一个 USB 设备而言，它至少具备两重身份：首先它是“USB”的，其次它是“自己”的。USB 设备是“USB”的，指它挂接在 USB 总线上，其必须完成 `usb_driver` 的初始化和注册；USB 设备是“自己”的，意味着本身可能是一个字符设备、tty 设备、网络设备等，因此，USB 设备驱动中也必须实现符合相应框架的代码。

USB 设备驱动的自身设备驱动部分的读写等操作流程有其特殊性，即以 URB 来贯穿始终，一个 URB 的生命周期通常包含创建、初始化、提交，和被 USB 核心及 USB 主机传递及完成后回调函数被调用的过程，当然，在 URB 被驱动提交后，也可以被取消。

在 UDC 和 gadget 侧，UDC 关心底层的硬件操作，而 gadget 驱动则只是利用通用的 API，透过 `usb_request` 与底层 UDC 驱动交互。

LINUX

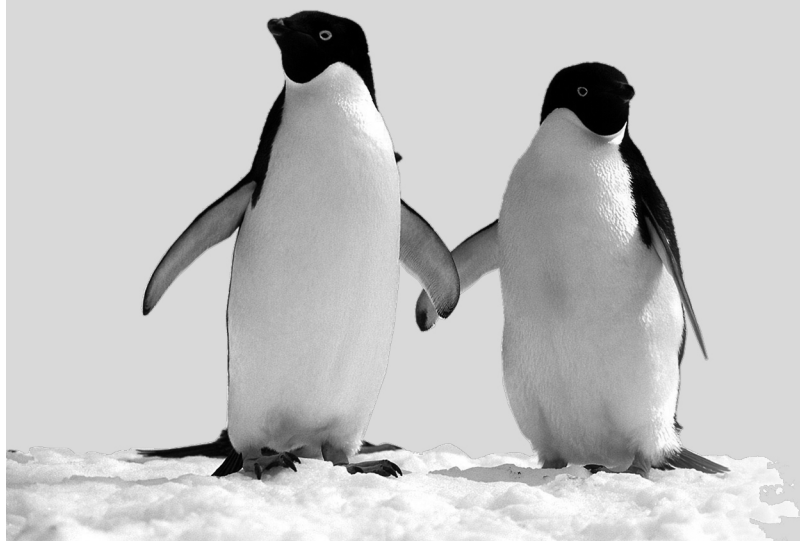
第21章

PCI 设备驱动

PCI 总线在一般的小型手持设备中不太可能用到，但是在工控和通信设备及其 PC 中却引领着潮流。在 Linux 系统中，PCI 设备驱动和 USB 设备驱动有共性，那就是其驱动都由总线相关部分和自身设备类型驱动两部分组成。

21.1 节讲解了 PCI 总线及其配置空间，给出了 PCI 总线在 Linux 内核中的数据结构。

PCI 设备驱动的 PCI 相关部分围绕着 `pci_driver` 结构体的成员函数展开，21.2 节讲解了 `pci_driver` 结构体及其成员函数的含义，并分析了 PCI 设备驱动的框架结构。





21.1 PCI 总线与配置空间

21.1.1 PCI 总线的 Linux 描述

PCI 是 CPU 和外围设备通信的高速传输总线。PCI 规范能够实现 32 位并行数据传输,工作频率为 33MHz 或 66MHz,最大吞吐率达 266MB/s。PCI 的衍生物包括 CardBus、mini-PCI、PCI-Express、cPCI 等。

从本书第 2 章的图 2.16 可以看出,PCI 总线体系结构是一种层次式的体系结构。在这种层次式体系结构中,PCI 桥设备占据着重要的地位,它将父总线与子总线连接在一起,从而使整个系统看起来像一颗倒置的树型结构。树的顶端是系统的 CPU,它通过一个较为特殊的 PCI 桥设备——Host/PCI 桥设备与根 PCI 总线连接起来。

作为一种特殊的 PCI 设备,PCI 桥包括以下几种。

- Host/PCI 桥:用于连接 CPU 与 PCI 根总线,第 1 个根总线的编号为 0。在 PC 中,内存控制器也通常被集成到 Host/PCI 桥设备芯片中,因此,Host/PCI 桥通常也被称为“北桥芯片组 (North Bridge Chipset)”。
- PCI/ISA 桥:用于连接旧的 ISA 总线。通常,PCI 中的类似 i8359A 中断控制器这样的设备也会被集成到 PCI/ISA 桥设备中,因此,PCI/ISA 桥通常也被称为“南桥芯片组 (South Bridge Chipset)”。
- PCI-to-PCI 桥:用于连接 PCI 主总线 (primary bus) 与次总线 (secondary bus)。PCI 桥所处的 PCI 总线称为“主总线”(即次总线的父总线),桥设备所连接的 PCI 总线称为“次总线”(即主总线的子总线)。

在 Linux 系统中,PCI 总线用 `pci_bus` 来描述,这个结构体记录了本 PCI 总线的信息以及本 PCI 总线的父总线、子总线、桥设备信息,这个结构体的定义如代码清单 21.1 所示。

代码清单 21.1 `pci_bus` 结构体

```
1 struct pci_bus {
2     struct list_head node; /* 链表元素 node */
3     struct pci_bus *parent; /* 指向该 PCI 总线的父总线,即 PCI 桥所在的总线 */
4     struct list_head children; /* 描述这条 PCI 总线的子总线链表的表头 */
5     struct list_head devices; /* 描述这条 PCI 总线的逻辑设备链表的表头 */
6     struct pci_dev *self; /* 指向引出这条 PCI 总线的桥设备的 pci_dev 结构 */
7     struct resource *resource[PCI_BUS_NUM_RESOURCES];
8     /* 指向应路由到这条 PCI 总线的地址空间资源 */
9
10    struct pci_ops *ops; /* 这条 PCI 总线所使用的配置空间访问函数 */
11    void *sysdata; /* 指向系统特定的扩展数据 */
12    struct proc_dir_entry *procdir; /* 该 PCI 总线在 /proc/bus/pci 中对应的目录项 */
13
14    unsigned char number; /* 这条 PCI 总线的总线编号 */
15    unsigned char primary; /* 桥设备的主总线 */
16    unsigned char secondary; /* PCI 总线的桥设备的次总线号 */
17    unsigned char subordinate; /* PCI 总线的下属 PCI 总线的总线编号最大值 */
18 }
```

```
19 char name[48];
20
21 unsigned short bridge_ctl;
22 pci_bus_flags_t bus_flags;
23 struct device *bridge;
24 struct class_device class_dev;
25 struct bin_attribute *legacy_io;
26 struct bin_attribute *legacy_mem;
27 unsigned int      is_added:1;
28 };
```

假定一个如图 21.1 所示的 PCI 总线系统，根总线 0 上有一个 PCI 桥，它引出子总线 Bus 1，Bus 1 上又有一个 PCI 桥引出 Bus 2。

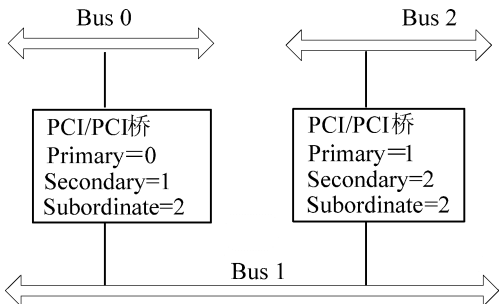


图 21.1 示例 PCI 总线系统

在上图中，Bus 0 总线的 pci_bus 结构体中的 number、primary、secondary 都应该为 0，因为它是通过 Host/PCI 桥引出的根总线；Bus 1 总线的 pci_bus 结构体中的 number 和 secondary 都为 1，但是它的 primary 应该为 0；Bus 2 总线的 pci_bus 结构体中的 number 和 secondary 都应该为 2，而其 primary 则应该等于 1。这 3 条总线的 subordinate 值都应该等于 2。

系统中当前存在的所有根总线都通过其 pci_bus 结构体中的 node 成员链接成一条全局的根总线链表，其表头由 list 类型的全局变量 pci_root_buses 来描述。而根总线下面的所有下级总线则都通过其 pci_bus 结构体中的 node 成员链接到其父总线的 children 链表中。这样，通过这两种 PCI 总线链表，Linux 内核就将所有的 pci_bus 结构体以一种倒置树的方式组织起来。假定对于如图 21.2 所示的多根 PCI 总线体系结构，它所对应的总线链表结构将如图 21.3 所示。

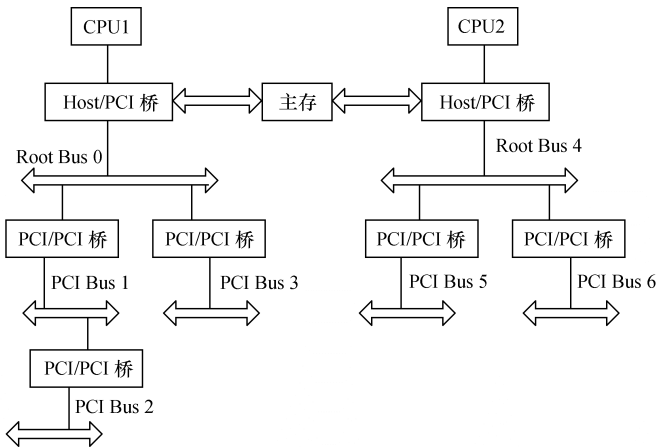


图 21.2 多根 PCI 总线体系结构

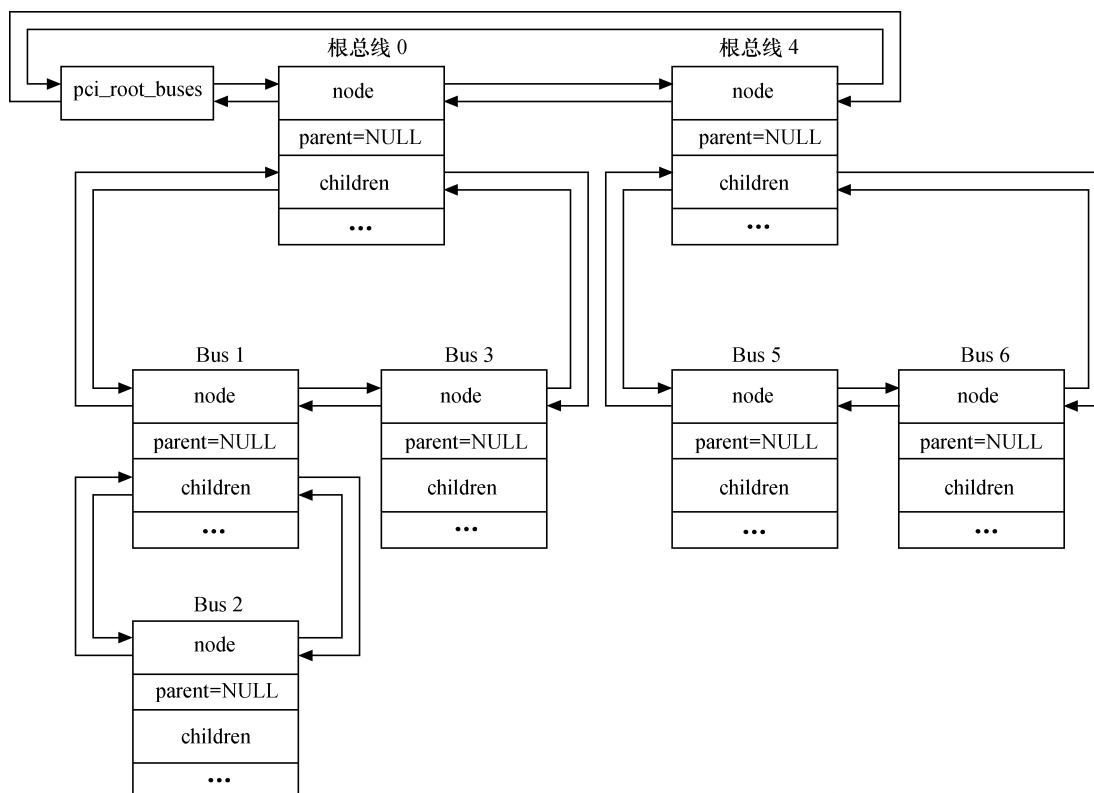


图 21.3 PCI 总线链表

21.1.2 PCI 设备的 Linux 描述

在 Linux 系统中, 所有种类的 PCI 设备都可以用 `pci_dev` 结构体来描述, 由于一个 PCI 接口卡上可能包含多个功能模块, 每个功能被当作一个独立的逻辑设备, 因此, 每一个 PCI 功能, 即 PCI 逻辑设备都唯一地对应一个 `pci_dev` 设备描述符, 该结构体的定义如代码清单 21.2 所示。

代码清单 21.2 `pci_dev` 结构体

```
1 struct pci_dev {
2     struct list_head bus_list;
3     struct pci_bus *bus; /* 这个 PCI 设备所在的 PCI 总线的 pci_bus 结构 */
4     struct pci_bus *subordinate; /* 指向这个 PCI 设备所桥接的下级总线 */
5
6     void *sysdata; /* 指向一片特定于系统的扩展数据 */
7     struct proc_dir_entry *procent; /* 该 PCI 设备在 /proc/bus/pci 中对应的目录项 */
8     struct pci_slot *slot; /* 设备位于的物理插槽 */
9     unsigned int devfn; /* 这个 PCI 设备的设备功能号 */
10    unsigned short vendor; /* PCI 设备的厂商 ID */
11    unsigned short device; /* PCI 设备的设备 ID */
12    unsigned short subsystem_vendor; /* PCI 设备的子系统厂商 ID */
13    unsigned short subsystem_device; /* PCI 设备的子系统设备 ID */
14    unsigned int class; /* 32 位的无符号整数, 表示该 PCI 设备的类别,
15        bit [7:0] 为编程接口, bit [15:8] 为子类别代码, bit [23:16]
16        为基类别代码, bit [31:24] 无意义 */
17    u8 hdr_type; /* PCI 配置空间头部的类型 */
18}
```



```

18  u8 pcie_type;      /* PCI-E 设备/端口类型 */
19  u8 rom_base_reg; /* 表示 PCI 配置空间中的 ROM 基地址寄存器在 PCI 配置空间中的位置 */
20  u8 pin; /* 中断引脚 */
21  struct pci_driver *driver; /* 指向这个 PCI 设备所对应的驱动 pci_driver 结构 */
22  u64 dma_mask; /* 该设备支持的总线地址位掩码, 通常是 0xffffffff */
23  struct device_dma_parameters dma_parms;
24  pci_power_t current_state; /* 当前的操作状态 */
25  int pm_cap;
26  unsigned int pme_support:5;
27  unsigned int d1_support:1; /* 支持 low power 状态 D1 ? */
28  unsigned int d2_support:1; /* 支持 low power 状态 D2? */
29  unsigned int no_d1d2:1; /* 仅允许 D0 和 D3 ? */
30  pci_channel_state_t error_state;
31  struct device dev; /* 通用的设备接口 */
32  int cfg_size; /* 配置空间大小 */
33
34  unsigned int irq;
35  struct resource resource[DEVICE_COUNT_RESOURCE];
36      /*表示该设备可能用到的资源, 包括:
37      I/O 端口区域、设备内存地址区域以及扩展 ROM 地址区域 */
38
39  unsigned int transparent: 1; /* 透明 PCI 桥 ? */
40  unsigned int multifunction: 1; /* 多功能设备 ? */
41  /* 跟踪设备状态 */
42  unsigned int is_added:1;
43  unsigned int is_busmaster: 1; /* 设备是主设备? */
44  unsigned int no_msi: 1; /* 设备可不使用 msi? */
45  unsigned int block_ucfg_access:1; /* 不允许用户空间访问配置空间 ? */
46  ...
47  u32 saved_config_space[16]; /* 挂起时保存的配置空间 */
48  struct bin_attribute *rom_attr; /* sysfs ROM 入口的属性描述 */
49  int rom_attr_enabled;
50  struct bin_attribute *res_attr[DEVICE_COUNT_RESOURCE]; /*资源的 sysfs 文件*/
51  ...
52 };

```

21.1.3 PCI 配置空间访问

PCI 有 3 种地址空间: PCI I/O 空间、PCI 内存地址空间和 PCI 配置空间。CPU 可以访问所有的地址空间, 其中 PCI I/O 空间和 PCI 内存地址空间由设备驱动程序使用。PCI 支持自动配置设备, 与旧的 ISA 驱动程序不一样, PCI 驱动程序不需要实现复杂的检测逻辑。启动时, BIOS 或内核自身会遍历 PCI 总线并分配资源, 如中断优先级和 I/O 基址。设备驱动程序通过 PCI 配置空间来找到资源分配情况。

PCI 规范定义了 3 种类型的 PCI 配置空间头部, 其中 type 0 用于标准的 PCI 设备, type 1 用于 PCI 桥, type 2 用于 PCI CardBus 桥。如图 2.17 所示, 不管是哪一种类型的配置空间头部, 其前 16 个字节的格式都是相同的, `/include/linux/pci_regs.h` 文件中定义了 PCI 配置空间头部, 如代码清单 21.3 所示。

代码清单 21.3 PCI 配置空间头部寄存器定义

```

1  #define PCI_VENDOR_ID      0x00 /* 16 位厂商 ID */
2  #define PCI_DEVICE_ID      0x02 /* 16 位设备 ID */
3
4  /* PCI 命令寄存器 */

```



```
5 #define PCI_COMMAND          0x04    /* 16 位 */
6 #define PCI_COMMAND_IO       0x1     /* 使能设备响应对 I/O 空间的访问 */
7 #define PCI_COMMAND_MEMORY   0x2     /* 使能设备响应对存储空间的访问 */
8 #define PCI_COMMAND_MASTER   0x4     /* 使能总线主模式 */
9 #define PCI_COMMAND_SPECIAL  0x8     /* 使能设备响应特殊周期 */
10 #define PCI_COMMAND_INVALIDATE 0x10  /* 使用 PCI 内存写无效事务 */
11 #define PCI_COMMAND_VGA_PALETTE 0x20 /* 使能 VGA 调色板侦测 */
12 #define PCI_COMMAND_PARITY   0x40    /* 使能奇偶校验 */
13 #define PCI_COMMAND_WAIT     0x80    /* 使能地址/数据步进 */
14 #define PCI_COMMAND_SERR     0x100   /* 使能 SERR */
15 #define PCI_COMMAND_FAST_BACK 0x200  /* 使能背靠背写 */
16 #define PCI_COMMAND_INTX_DISABLE 0x400 /* 禁止中断竞争 */
17
18 /* PCI 状态寄存器 */
19 #define PCI_STATUS           0x06    /* 16 位 */
20 #define PCI_STATUS_CAP_LIST   0x10    /* 支持的能力列表 */
21 #define PCI_STATUS_66MHz     0x20    /* 支持 PCI 2.1 66MHz */
22 #define PCI_STATUS_UDF       0x40    /* 支持用户定义的特征 */
23 #define PCI_STATUS_FAST_BACK  0x80    /* 快速背靠背操作 */
24 #define PCI_STATUS_PARITY    0x100   /* 侦测到奇偶校验错 */
25 #define PCI_STATUS_DEVSEL_MASK 0x600  /* DEVSEL 定时 */
26 #define PCI_STATUS_DEVSEL_FAST 0x000
27 #define PCI_STATUS_DEVSEL_MEDIUM 0x200
28 #define PCI_STATUS_DEVSEL_SLOW 0x400
29 #define PCI_STATUS_SIG_TARGET_ABORT 0x800 /* 目标设备异常 */
30 #define PCI_STATUS_REC_TARGET_ABORT 0x1000 /* 主设备确认 */
31 #define PCI_STATUS_REC_MASTER_ABORT 0x2000 /* 主设备异常 */
32 #define PCI_STATUS_SIG_SYSTEM_ERROR 0x4000 /* 驱动了 SERR */
33 #define PCI_STATUS_DETECTED_PARITY 0x8000 /* 奇偶校验错 */
34
35 /* 类代码寄存器和修订版本寄存器 */
36 #define PCI_CLASS_REVISION    0x08    /* 高 24 位为类码, 低 8 位为修订版本 */
37 #define PCI_REVISION_ID      0x08    /* 修订号 */
38 #define PCI_CLASS_PROG       0x09    /* 编程接口 */
39 #define PCI_CLASS_DEVICE     0x0a    /* 设备类 */
40 #define PCI_CACHE_LINE_SIZE  0x0c    /* 8 位 */
41 #define PCI_LATENCY_TIMER    0x0d    /* 8 位 */
42
43 /* PCI 头类型 */
44 #define PCI_HEADER_TYPE      0x0e    /* 8 位头类型 */
45 #define PCI_HEADER_TYPE_NORMAL 0
46 #define PCI_HEADER_TYPE_BRIDGE 1
47 #define PCI_HEADER_TYPE_CARDBUS 2
48
49 /* 表示配置空间头部中的 Built-In Self-Test 寄存器在配置空间中的字节地址索引 */
50 #define PCI_BIST             0x0f    /* 8 位 */
51 #define PCI_BIST_CODE_MASK   0x0f    /* 完成代码 */
52 #define PCI_BIST_START       0x40    /* 用于启动 BIST */
53 #define PCI_BIST_CAPABLE     0x80    /* 设备是否支持 BIST? */
```

紧接着前 16 个字节的寄存器为基地址寄存器 0~基地址寄存器 5, 其中, `PCI_BASE_ADDRESS_2~5` 仅对标准 PCI 设备的 0 类型配置空间头部有意义, 而 `PCI_BASE_ADDRESS_0~1` 则适用于 0 类型和 1 类型配置空间头部。

基地址寄存器中的 `bit [0]` 的值决定了这个基地址寄存器所指定的地址范围是在 I/O 空间还

是在内存映射空间内进行译码。当基地址寄存器所指定的地址范围位于内存映射空间中时，其 bit [2:1] 表示内存地址的类型，bit [3] 表示内存范围是否为可预取（Prefetchable）的内存。

1 类型配置空间头部适用于 PCI-PCI 桥设备，其基地址寄存器 0 与基地址寄存器 1 可以用来指定桥设备本身可能要用到的地址范围，而后 40 个字节（0x18~0x39）则被用来配置桥设备的主、次编号以及地址过滤窗口等信息。

pci_bus 结构体中的 pci_ops 类型成员指针 ops 指向该 PCI 总线所使用的配置空间访问操作的具体实现，pci_ops 结构体的定义如代码清单 21.4 所示。

代码清单 21.4 pci_ops 结构体

```
1 struct pci_ops {
2     int(*read)(struct pci_bus *bus, unsigned int devfn, int where, int size, u32
3         *val); /* 读配置空间 */
4     int(*write)(struct pci_bus *bus, unsigned int devfn, int where, int size, u32
5         val); /* 写配置空间 */
6 };
```

read()和 write()成员函数中的 size 表示访问的是字节、2 字节还是 4 字节，对于 write()而言，val 是要写入的值；对于 read()而言，val 是要返回的读取到的值的指针。通过 bus 参数的成员以及 devfn 可以定位相应 PCI 总线上相应 PCI 逻辑设备的配置空间。在 Linux 设备驱动中，可用如下的一组函数来访问配置空间：

```
inline int pci_read_config_byte(struct pci_dev *dev, int where, u8 *val);
inline int pci_read_config_word(struct pci_dev *dev, int where, u16 *val);
inline int pci_read_config_dword(struct pci_dev *dev, int where, u32 *val);
inline int pci_write_config_byte(struct pci_dev *dev, int where, u8 val);
inline int pci_write_config_word(struct pci_dev *dev, int where, u16 val);
inline int pci_write_config_dword(struct pci_dev *dev, int where, u32 val);
```

上述函数只是对如下函数进行调用：

```
int pci_bus_read_config_byte (struct pci_bus *bus, unsigned int devfn, int where, u8 *val);
/* 读字节 */
int pci_bus_read_config_word (struct pci_bus *bus, unsigned int devfn, int where, u16
*val); /* 读字 */
int pci_bus_read_config_dword (struct pci_bus *bus, unsigned int devfn, int where, u32
*val); /* 读双字 */
int pci_bus_write_config_byte (struct pci_bus *bus, unsigned int devfn, int where, u8
val); /* 写字节 */
int pci_bus_write_config_word (struct pci_bus *bus, unsigned int devfn, int where, u16
val); /* 写字 */
int pci_bus_write_config_dword (struct pci_bus *bus, unsigned int devfn, int where, u32
val); /* 写双字 */
```

最后，我们来看一下 PCI 总线、设备与驱动在 /proc 和 /sysfs 文件系统描述。首先，通过查看 /proc/bus/pci 中的文件，可以获得系统连接的 PCI 设备的基本信息描述。在本书配套虚拟机 Linux 上的 /proc/bus/pci 目录下的树型结构如下：

```
/proc/bus/pci
|-- 00
|   |-- 00.0
|   |-- 01.0
|   |-- 01.1
|   |-- 02.0
|   |-- 03.0
```



```
| |-- 04.0
| |-- 05.0
| |-- 06.0
| |-- 07.0
| `-- 0b.0
|-- devices
```

```
1 directory, 11 files
```

sysfs 文件系统/sys/bus/pci 目录中也给出了系统中总线上挂接的设备及驱动信息, 该目录下的树型结构如下:

```
/sys/bus/pci
|-- devices
| |-- 0000:00:00.0 → ../../../../devices/pci0000:00/0000:00:00.0
| |-- 0000:00:01.0 → ../../../../devices/pci0000:00/0000:00:01.0
| |-- 0000:00:01.1 → ../../../../devices/pci0000:00/0000:00:01.1
| |-- 0000:00:02.0 → ../../../../devices/pci0000:00/0000:00:02.0
| |-- 0000:00:03.0 → ../../../../devices/pci0000:00/0000:00:03.0
| |-- 0000:00:04.0 → ../../../../devices/pci0000:00/0000:00:04.0
| |-- 0000:00:05.0 → ../../../../devices/pci0000:00/0000:00:05.0
| |-- 0000:00:06.0 → ../../../../devices/pci0000:00/0000:00:06.0
| |-- 0000:00:07.0 → ../../../../devices/pci0000:00/0000:00:07.0
| `-- 0000:00:0b.0 → ../../../../devices/pci0000:00/0000:00:0b.0
|-- drivers
| |-- Intel ICH
| | |-- 0000:00:05.0 → ../../../../devices/pci0000:00/0000:00:05.0
| | |-- bind
| | |-- module → ../../../../module/snd_intel8x0
| | |-- new_id
| | |-- uevent
| | `-- unbind
| ...
| |-- pcnet32
| | |-- 0000:00:03.0 → ../../../../devices/pci0000:00/0000:00:03.0
| | |-- bind
| | |-- module → ../../../../module/pcnet32
| | |-- new_id
| | |-- uevent
| | `-- unbind
|
|-- drivers_autoprobe
|-- drivers_probe
|-- slots
`-- uevent
```

```
89 directories, 239 files
```

此外, pciutils (PCI 工具) 中的 lspci 工具会分析 /proc/bus/pci 中的文件, 从而可被用户用于查看系统中 PCI 设备的描述信息, 例如在本书配套虚拟机上运行 lspci 的结果为:

```
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371AB/EB/MB PIIX4 IDE (rev 01)
00:02.0 VGA compatible controller: InnoTek Systemberatung GmbH VirtualBox Graphics Adapter
00:03.0 Ethernet controller: Advanced Micro Devices [AMD] 79c970 [PCnet32 LANCE] (rev
```

```

40)
    00:04.0 System peripheral: InnoTek Systemberatung GmbH VirtualBox Guest Service
    00:05.0 Multimedia audio controller: Intel Corporation 82801AA AC'97 Audio Controller
(rev 01)
    00:06.0 USB Controller: Apple Computer Inc. KeyLargo/Intrepid USB
    00:07.0 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 08)
    00:0b.0 USB Controller: Intel Corporation 82801FB/FBM/FR/FW/FRW (ICH6 Family) USB2 EHCI
Controller
  
```

21.1.4 PCI DMA 相关的 API

内核中定义了一组专门针对 PCI 设备的 DMA 操作接口，这些 API 的原型和作用与第 11 章介绍的通用 DMA API 非常相似，主要包括如下。

- 设置 DMA 缓冲区掩码。

```
int pci_set_dma_mask(struct pci_dev *dev, u64 mask);
```

- 一致性 DMA 缓冲区分配/释放。

```

void *pci_alloc_consistent(struct pci_dev *pdev, size_t size, dma_addr_t *dma_handle);
void pci_free_consistent(struct pci_dev *hwdev, size_t size, void *vaddr, dma_addr_t
dma_handle);
  
```

- 流式 DMA 缓冲区映射/去映射。

```

dma_addr_t pci_map_single(struct pci_dev *pdev, void *ptr, size_t size, int direction);

int pci_map_sg(struct pci_dev *pdev, struct scatterlist *sgl, int num_entries, int
direction);

void pci_unmap_single(struct pci_dev *pdev, dma_addr_t dma_addr, size_t size,
int direction);

void pci_unmap_sg(struct pci_dev *pdev, struct scatterlist *sg, int nents, int
direction);
  
```

这些 API 的用法与第 11 章中介绍的 `dma_alloc_consistent()`、`dma_map_single()`、`dma_map_sg()` 相似，只是以 `pci_` 开头的 API 用于 PCI 设备驱动。

21.1.5 PCI 设备驱动其他常用 API

除了 DMA API 外，在 PCI 设备驱动中其他常用的函数（或宏）如下所示。

- 获取 I/O 或内存资源。

```

#define pci_resource_start(dev, bar) ((dev)→resource[(bar)].start)
#define pci_resource_end(dev, bar) ((dev)→resource[(bar)].end)
#define pci_resource_flags(dev, bar) ((dev)→resource[(bar)].flags)
#define pci_resource_len(dev, bar) \
    ((pci_resource_start((dev), (bar)) == 0 && \
      pci_resource_end((dev), (bar)) == \
        pci_resource_start((dev), (bar))) ? 0 : \
      \
      (pci_resource_end((dev), (bar)) - \
        pci_resource_start((dev), (bar)) + 1))
  
```

- 申请/释放 I/O 或内存资源。

```

int pci_request_regions(struct pci_dev *pdev, const char *res_name);
void pci_release_regions(struct pci_dev *pdev);
  
```

- 获取/设置驱动私有数据。



```
void *pci_get_drvdata (struct pci_dev *pdev);  
void pci_set_drvdata (struct pci_dev *pdev, void *data);
```

- 使能/禁止 PCI 设备。

```
int pci_enable_device(struct pci_dev *dev);  
void pci_disable_device(struct pci_dev *dev);
```

- 设置为总线主 DMA。

```
void pci_set_master(struct pci_dev *dev);
```

- 寻找指定总线指定槽位的 PCI 设备。

```
struct pci_dev *pci_find_slot (unsigned int bus, unsigned int devfn);
```

- 设置 PCI 能量管理状态 (0=D0 ... 3=D3)。

```
int pci_set_power_state(struct pci_dev *dev, pci_power_t state);
```

- 在设备的能力表中找出指定的能力。

```
int pci_find_capability (struct pci_dev *dev, int cap);
```

- 启用设备内存写无效事务。

```
int pci_set_mwi(struct pci_dev *dev);
```

- 禁用设备内存写无效事务。

```
void pci_clear_mwi(struct pci_dev *dev);
```

21.2 PCI 设备驱动结构

21.2.1 PCI 设备驱动的组成

从本质上讲 PCI 只是一种总线，具体的 PCI 设备可以是字符设备、网络设备、USB 主机控制器等，因此，一个通过 PCI 总线与系统连接的设备的驱动至少包含以下两部分内容。

- PCI 驱动。
- 设备本身的驱动。

PCI 驱动只是为了辅助设备本身的驱动，它不是目的，而是手段。例如，对于通过 PCI 总线与系统连接的字符设备，则驱动中除了要实现 PCI 驱动部分外，其主体仍然是设备作为字符设备本身的驱动，即实现 `file_operations` 成员函数并注册 `cdev`。分析 Linux 内核可知，在 `/drivers/block/`、`/drivers/atm/`、`/drivers/char/`、`/drivers/i2c/`、`/drivers/ieee1394/`、`/drivers/media/`、`/drivers/mtd/`、`/drivers/net/`、`/drivers/serial/`、`/drivers/video/`、`/sound/` 等目录中均广泛分布着 PCI 设备驱动。

在 Linux 内核中，用 `pci_driver` 结构体来定义 PCI 驱动，该结构体中包含了 PCI 设备的探测/移除、挂起/恢复等函数，其定义如代码清单 21.5 所示。`pci_driver` 和前面说的 `platform_driver`、`i2c_driver`、`usb_driver` 的地位非常相似，都是起到挂接总线的作用。

代码清单 21.5 `pci_driver` 结构体

```
1 struct pci_driver {  
2     struct list_head node;  
3     char *name;  
4     struct module *owner;  
5     const struct pci_device_id *id_table; /*不能为 NULL，以便 probe 函数调用*/  
6     /* 新设备添加 */
```

```

7  int(*probe)(struct pci_dev *dev, const struct pci_device_id *id);
8  void(*remove)(struct pci_dev *dev); /* 设备移出 */
9  int(*suspend)(struct pci_dev *dev, pm_message_t state); /* 设备挂起 */
10 int (*suspend_late)(struct pci_dev *dev, pm_message_t state);
11 int (*resume_early)(struct pci_dev *dev);
12 int(*resume)(struct pci_dev *dev); /* 设备唤醒 */
13 void(*shutdown)(struct pci_dev *dev);
14 struct pm_ext_ops *pm;
15 struct pci_error_handlers *err_handler;
16 struct device_driver driver;
17 struct pci_dynids dynids;
18 };

```

对 `pci_driver` 的注册和注销通过如下函数来实现：

```

int pci_register_driver(struct pci_driver *driver);
void pci_unregister_driver(struct pci_driver *drv);

```

`pci_driver` 的 `probe()` 函数要完成 PCI 设备的初始化及其设备本身身份（字符、TTY、网络等）驱动的注册。当 Linux 内核启动并完成对所有 PCI 设备进行扫描、登录和分配资源等初始化操作的同时，会建立起系统中所有 PCI 设备的拓扑结构，`probe()` 函数将负责硬件的探测工作并保存配置信息。

下面以一个 PCI 接口字符设备为例，给出 PCI 设备驱动的完整模板。其一部分代码实现 `pci_driver` 成员函数，一部分代码实现字符设备的 `file_operations` 成员函数，如代码清单 21.6 所示。

代码清单 21.6 PCI 设备驱动的程序模板

```

1  /* 指明该驱动程序适用于哪一些 PCI 设备 */
2  static struct pci_device_id xxx_pci_tbl [] __initdata = {
3      {PCI_VENDOR_ID_DEMO, PCI_DEVICE_ID_DEMO,
4       PCI_ANY_ID, PCI_ANY_ID, 0, 0, DEMO},
5      {0,}
6  };
7  MODULE_DEVICE_TABLE(pci, xxx_pci_tbl);
8
9  /* 中断处理函数 */
10 static void xxx_interrupt(int irq, void *dev_id, struct pt_regs *regs)
11 {
12     ...
13 }
14
15 /* 字符设备 file_operations open 成员函数 */
16 static int xxx_open(struct inode *inode, struct file *file)
17 {
18     /* 申请中断，注册中断处理程序 */
19     request_irq(xxx_irq, &xxx_interrupt, ...);
20     ...
21 }
22
23 /* 字符设备 file_operations ioctl 成员函数 */
24 static int xxx_ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)
25 {
26     ...
27 }
28

```



```
29 /* 字符设备 file_operations read、write、mmap 等成员函数 */
30
31 /* 设备文件操作接口 */
32 static struct file_operations xxx_fops = {
33     owner:      THIS_MODULE,          /* xxx_fops 所属的设备模块 */
34     read:       xxx_read,             /* 读设备操作 */
35     write:      xxx_write,            /* 写设备操作 */
36     ioctl:      xxx_ioctl,           /* 控制设备操作 */
37     mmap:       xxx_mmap,             /* 内存重映射操作 */
38     open:       xxx_open,             /* 打开设备操作 */
39     release:    xxx_release           /* 释放设备操作 */
40 };
41
42 /* pci_driver 的 probe 成员函数 */
43 static int __init xxx_probe(struct pci_dev *pci_dev, const struct pci_device_id *pci_id)
44 {
45     pci_enable_device(pci_dev); /* 启动 PCI 设备 */
46
47     /* 读取 PCI 配置信息 */
48     iobase = pci_resource_start (pci_dev, 1);
49     ...
50
51     pci_set_master(pci_dev); // 设置成总线主 DMA 模式
52
53     pci_request_regions(pci_dev); /* 申请 I/O 资源 */
54
55     /* 注册字符设备 */
56     cdev_init(&xxx_cdev, &xxx_fops);
57     register_chrdev_region(xxx_dev_no, 1, ...);
58     cdev_add(&xxx_cdev);
59
60     return 0;
61 }
62
63 /* pci_driver 的 remove 成员函数 */
64 static int __init xxx_release(struct pci_dev *pdev)
65 {
66     pci_release_regions(pdev); /* 释放 I/O 资源 */
67     pci_disable_device (pdev); /* 禁止 PCI 设备 */
68     unregister_chrdev_region(xxx_dev_no, 1); /* 释放占用的设备号 */
69     cdev_del(&xxx_dev.cdev); /* 注销字符设备 */
70     ...
71     return 0;
72 }
73
74 /* 设备模块信息 */
75 static struct pci_driver xxx_pci_driver = {
76     name:      xxx_MODULE_NAME,      /* 设备模块名称 */
77     id_table:   xxx_pci_tbl,          /* 能够驱动的设备列表 */
78     probe:     xxx_probe,             /* 查找并初始化设备 */
79     remove:    xxx_remove             /* 卸载设备模块 */
80 };
81
82 static int __init xxx_init_module (void)
83 {
```



```

84     pci_register_driver(&xxx_pci_driver);
85 }
86 static void __exit xxx_cleanup_module (void)
87 {
88     pci_unregister_driver(&xxx_pci_driver);
89 }
90 /*驱动模块加载函数 */
91 module_init(xxx_init_module);
92 /* 驱动模块卸载函数 */
93 module_exit(xxx_cleanup_module);

```

上述代码清单的第 1~7 行给出了本驱动所支持的 PCI 设备的列表，如同在 USB 设备驱动中定义 `usb_device_id` 结构体数组一样，在 PCI 设备驱动中，也需要定义一个 `pci_device_id` 结构体数组。`pci_device_id` 用于标识一个 PCI 设备。它的成员包括：厂商 ID、设备 ID、子厂商 ID、子设备 ID、类别、类别掩码（类可分为基类、子类）和私有数据。每一个 PCI 设备的驱动程序都有一个 `pci_device_id` 的数组，用于告诉 PCI 核心自己能够驱动哪些设备，`pci_device_id` 结构体的定义如代码清单 21.7 所示。

代码清单 21.7 `pci_device_id` 结构体

```

1 struct pci_device_id {
2     __u32 vendor, device;          /* 厂商和设备 ID 或 PCI_ANY_ID */
3     __u32 subvendor, subdevice;   /* 子厂商 ID 或 PCI_ANY_ID */
4     __u32 class, class_mask;      /* (类、子类、prog-if) 三元组 */
5     kernel_ulong_t driver_data;   /* 驱动私有数据 */
6 };

```

将代码清单 21.6 中的各个函数进行归类，可得出该驱动的组成，如图 21.4 所示。

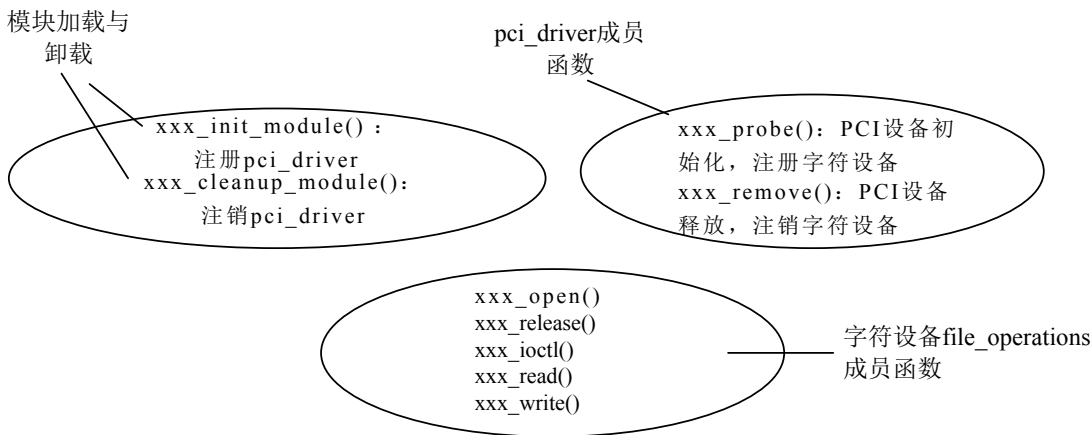


图 21.4 PCI 字符设备驱动的组成

图 21.5 所示的树中，树根是主机/PCI 桥，树叶是具体的 PCI 设备，树叶本身与树枝通过 `pci_driver` 连接，而树叶本身的驱动，读写、控制树叶则需要通过其树叶设备本身所属类设备驱动来完成。

由此我们看出，对于 USB、PCI 设备这种挂接在总线上的设备而言，USB、PCI 只是它们的“工作单位”，它们需要向“工作单位”注册（注册 `usb_driver`、`pci_driver`），并接收“工作单位”



的管理 (被调入 `probe()`、调出 `disconnect()/remove()`、放假 `suspend()/shutdown()`、继续上班 `resume()` 等), 但是其本身可能是一个工程师、一个前台或一个经理, 因此, 做好工程师、前台或经理是其主体工作, 这部分对应于字符设备驱动、tty 设备驱动、网络设备驱动等。

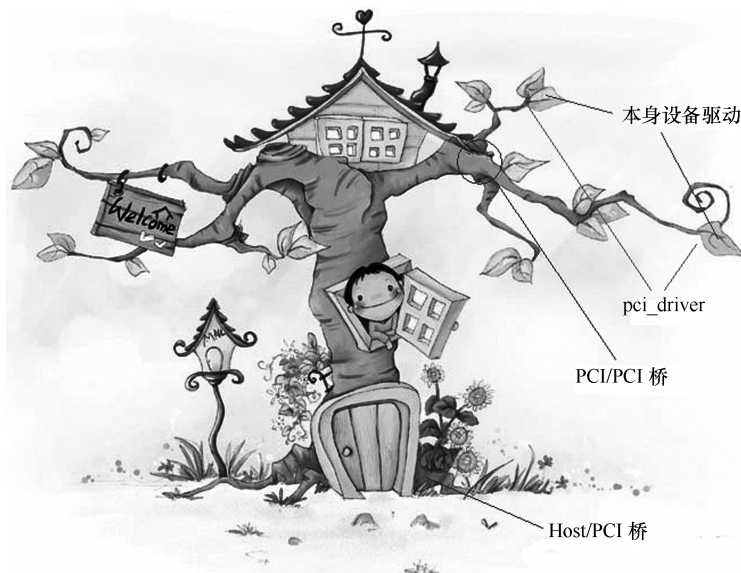


图 21.5 PCI 设备驱动的组成

21.2.2 实例：PCI 骨架程序

`drivers/net/pci-skeleton.c` 给出了一个 PCI 接口网络设备驱动程序的“骨架”, 其 `pci_driver` 中 `probe()` 成员函数 `netdrv_init_one()` 及其调用的 `netdrv_init_board()` 完成了 PCI 设备初始化及对应的网络设备注册工作, 代码清单 21.8 所示为这两个函数的实现。

代码清单 21.8 pci-skeleton 设备驱动的 `probe()` 函数

```
1 static int __devinit netdrv_init_one (struct pci_dev *pdev,  
2                                     const struct pci_device_id *ent)  
3 {  
4     struct net_device *dev = NULL;  
5     struct netdrv_private *tp;  
6     ...  
7     i = netdrv_init_board (pdev, &dev, &iaddr);  
8     ...  
9  
10    dev->open = netdrv_open;  
11    dev->hard_start_xmit = netdrv_start_xmit;  
12    dev->stop = netdrv_close;  
13    dev->set_multicast_list = netdrv_set_rx_mode;  
14    dev->do_ioctl = netdrv_ioctl;  
15    dev->tx_timeout = netdrv_tx_timeout;  
16    dev->watchdog_timeo = TX_TIMEOUT;  
17  
18    dev->irq = pdev->irq;  
19    dev->base_addr = (unsigned long) iaddr;
```

```

20
21  tp = netdev_priv(dev);
22
23  ...
24  pci_set_drvdata(pdev, dev);
25  ...
26  return 0;
27 }
28
29 static int __devinit netdrv_init_board (struct pci_dev *pdev,
30                                         struct net_device **dev_out,
31                                         void **ioaddr_out)
32 {
33  ...
34  dev = alloc_etherdev (sizeof (*tp));
35  if (dev == NULL) {
36      dev_err(&pdev->dev, "unable to alloc new ethernet\n");
37      DPRINTK ("EXIT, returning -ENOMEM\n");
38      return -ENOMEM;
39  }
40  SET_NETDEV_DEV(dev, &pdev->dev);
41  tp = netdev_priv(dev);
42  ...
43 }

```

从上述代码可以看出，`probe()`函数中进行了网络设备的初始化和注册。`pci_driver` 的 `remove()` 成员函数完成相反的工作，即注销网络设备，如代码清单 21.9 所示。

代码清单 21.9 pci-skeleton 设备驱动的 `remove()` 函数

```

1 static void __devexit netdrv_remove_one (struct pci_dev *pdev)
2 {
3     struct net_device *dev = pci_get_drvdata (pdev);
4     struct netdrv_private *np;
5
6     ...
7     np = netdev_priv(dev);
8     assert (np != NULL);
9
10    unregister_netdev (dev);
11
12    #ifndef USE_IO_OPS
13        iounmap (np->mmio_addr);
14    #endif /* !USE_IO_OPS */
15
16    pci_release_regions (pdev);
17
18    free_netdev (dev);
19
20    pci_set_drvdata (pdev, NULL);
21
22    pci_disable_device (pdev);
23
24    DPRINTK ("EXIT\n");
25 }

```

`/drivers/net/pci-skeleton.c` 中定义的 `pci_device_id` 结构体数组及 `MODULE_DEVICE_TABLE`



导出代码如清单 21.10 所示。

代码清单 21.10 PCI 设备驱动的 pci_device_id 数组及 MODULE_DEVICE_TABLE

```
1 static struct pci_device_id netdrv_pci_tbl[] = {
2     {0x10ec, 0x8139, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
3     {0x10ec, 0x8138, PCI_ANY_ID, PCI_ANY_ID, 0, 0, NETDRV_CB },
4     {0x1113, 0x1211, PCI_ANY_ID, PCI_ANY_ID, 0, 0, SMC1211TX },
5     /* {0x1113, 0x1211, PCI_ANY_ID, PCI_ANY_ID, 0, 0, MPX5030 },*/
6     {0x1500, 0x1360, PCI_ANY_ID, PCI_ANY_ID, 0, 0, DELTA8139 },
7     {0x4033, 0x1360, PCI_ANY_ID, PCI_ANY_ID, 0, 0, ADDTRON8139 },
8     {0,}
9 };
10 MODULE_DEVICE_TABLE (pci, netdrv_pci_tbl);
```

除此之外，pci-skeleton 的主体即是完成网络设备驱动相关的工作，完全符合本书第 6 章的模板。

21.3 总结

PCI 设备驱动只是字符设备、tty 设备、网络设备、音频设备等与系统的一个接口，因此，驱动将由两部分组成，一部分是 PCI 相关部分，另一部分是设备本身所属类驱动。PCI 驱动的核心数据结构是 pci_driver，在 probe() 成员函数中将申请资源并注册对应的字符设备、tty 设备、网络设备、音频设备等，而 remove() 成员函数中将释放资源并注销对应的字符设备、tty 设备、网络设备、音频设备等。

LINUX

第22章 Linux 设备驱动的调试

“工欲善其事，必先利其器”，为了方便进行 Linux 设备驱动的开发和调试，建立良好的开发环境很重要，包括实验室环境建设、使用必要的工具软件以及掌握常用的调试技巧等。

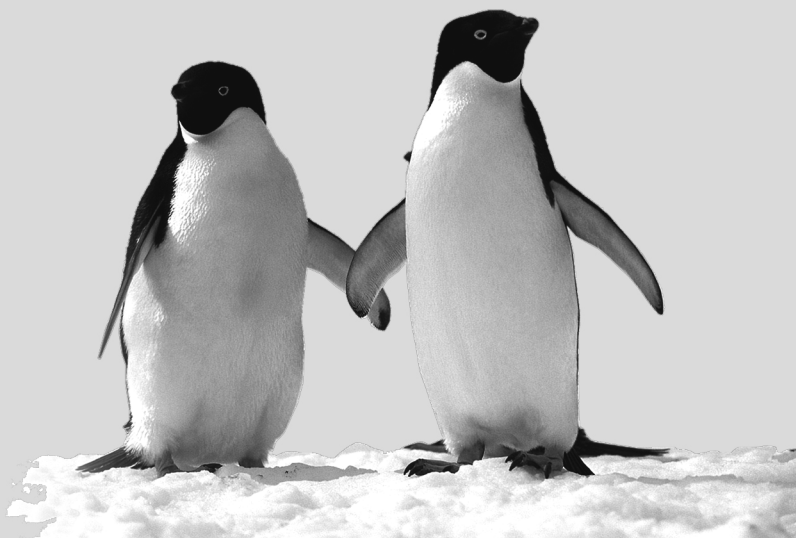
22.1 节介绍 Linux 开发环境的建设，包括实验室配置、工具链、串口工具等。

22.2 节讲解了 Linux 下调试器 gdb 的基本用法和技巧。

22.3 节讲解了 Linux 内核的调试方法，22.4~22.9 节对 22.3 节的概述展开讲解，分别讲解了 Linux 内核调试用到的 `printk()`、`/proc`、`oops`、监视工具，`kcore`、`kdb` 和 `kgdb`，以及使用仿真器进行调试的方法。

22.10 节讲解了 Linux 应用程序的调试方法，驱动工程师往往需要编写用户空间的应用程序对自身编写的驱动进行验证和测试，因此，掌握应用程序调试方法对驱动工程师而言也是必须的。

22.11 节讲解了 Linux 常用的一些稳定性、性能分析和调优工具。



22.1 Linux 开发环境建设

22.1.1 实验室建设

在公司或学校的实验室中，PC 的性能一般来说不会太高，用 PC 来编译 Linux 内核和模块的速度总是受限。相反地，服务器的资源相对比较充分，CPU 以及磁盘性能都较高，因此在服务器上进行内核、驱动及应用程序的编译开发都将更加快捷，而且使用服务器更便于统一管理实验室内的所有开发者。图 22.1 所示为一种常见的小型 Linux 实验室环境。

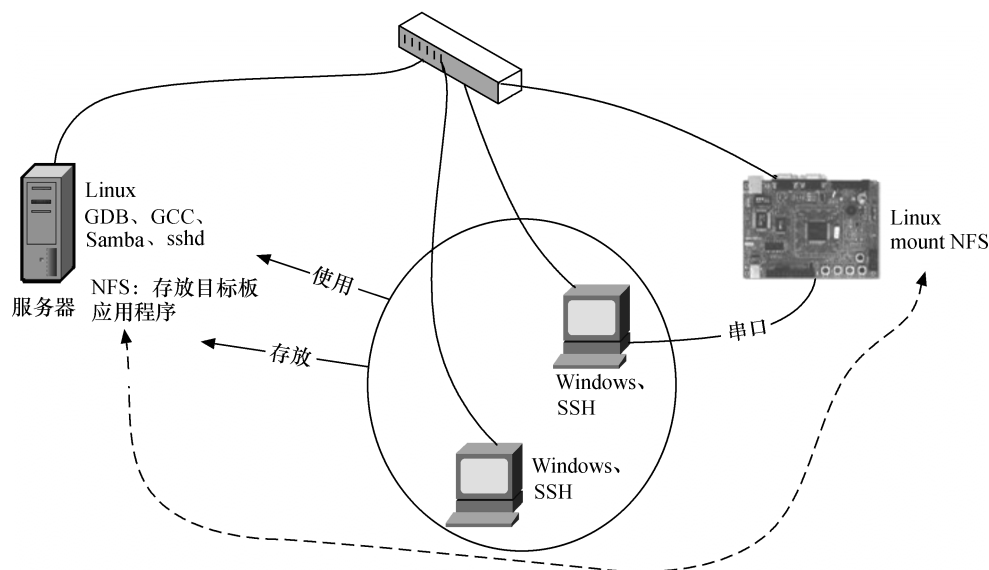


图 22.1 Linux 实验室环境

Linux 服务器上启动了 Samba 和 sshd 进程，各工程师在自己的 Windows 或 Linux 客户机上通过 SSH 用自己的用户名和密码登录服务器便可以使用服务器上的 GCC、GDB 等软件（Windows 下 SSH Secure Shell 界面如图 22.2 所示）。同时，SSH 软件提供了类似于 FTP 的文件共享功能（Windows 下 SSH Secure File Transfer 界面如图 22.3 所示），方便在客户端和服务端复制文件。

目标板、服务器和客户端全部通过交换机连接，同时客户端连接目标板的串口作为控制台。在调试 Linux 应用程序时，目标板 erver 与调试用的 GDB，目标板与服务器的 NFS 挂接都借助网络通信解决。编写完成的应用程序或内核模块可直接存放在服务器的 NFS 服务目录内，而该目录可被目标板上的 Linux 系统 mount 到本身的一个目录内。

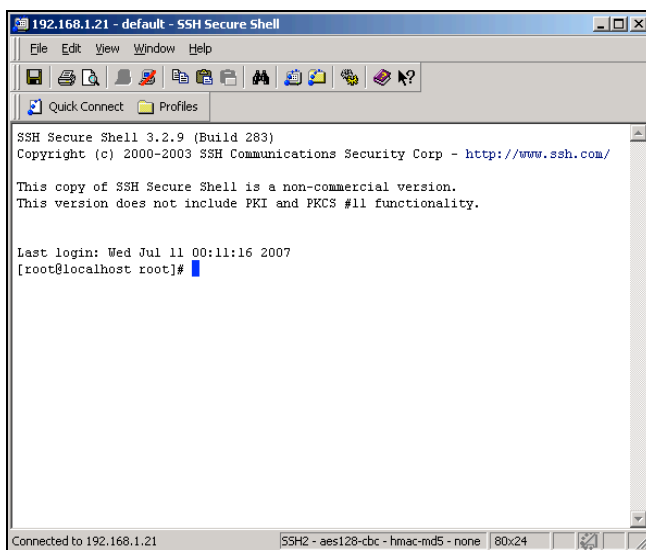


图 22.2 SSH Secure Shell

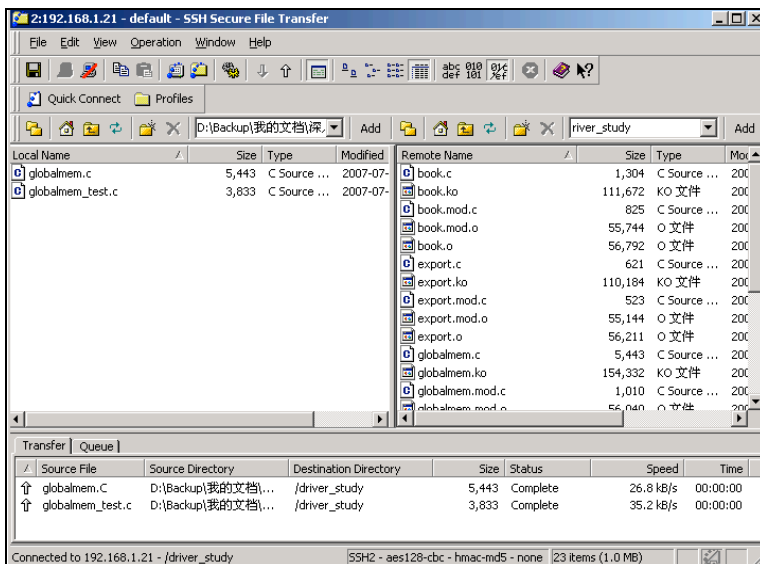


图 22.3 SSH Secure File Transfer

22.1.2 工具链

为了编译、连接并调试 Linux 应用程序和内核，我们首先需要建立针对目标板处理器的 GNU 工具链。

GNU 工具链有力地支撑了 Linux 系统的发展，由于它可被看作许多嵌入式处理器的一个交叉编译器，所以在嵌入式软件开发中相当流行，其支持包括 ARM、StrongARM、XScale、PowerPC、MIPS、68K/ColdFire、Intel x86/IA-32、Intel i960、Hitachi SH 在内的多种体系结构。GNU 工具链中大多数有用的工具主要集中于以下几个源代码包中。

- GCC 包，主要包括 gcc (C 编译器)、g++ (C++编译器)、cpp (C 预处理器)。
- Binutils (binary utilities)包，主要包括 as (汇编程序)、ld (连接器)、objcopy (目标文件翻译器，用于从连接器输出中创建一个 ROM 映像)、objdump (目标文件阅读器，用于反汇编目标文件)。
- glibc/uclibc/newlib，提供系统调用和基本函数的 C 库，比如 open()、malloc()、printf()等。
- Make，主要包括 make 工具。
- Debugger，主要包括 gdb (源代码调试器)。

上述源代码包都可以从 gnu FTP 站点上直接下载，如为了建立 ARM Linux 的 GNU 工具链，我们需下载 newlib、binutils、gcc 和 gdb 代码包。在解压缩、针对特定处理器配置、编译并安装这样软件包后，便可以使用它们进行交叉编译与开发。配置、编译与安装的所需执行的命令步骤如下：

```
(1)cd [binutils-build]
(2)[binutils-source]/configure --target=arm-elf --prefix=[toolchain-prefix] --enable-
interwork --enable-multilib
(3)make all install
(4)export PATH="$PATH:[toolchain-prefix]/bin"
(5)cd [gcc-build]
(6)[gcc-source]/configure --target=arm-elf --prefix=[toolchain-prefix] --enable-interwork --
enable-multilib --enable-languages="c,c++" --with-newlib --with-headers=[newlib-source]/newlib/
libc/include
(7)make all-gcc install-gcc
(8)cd [newlib-build]
(9)[newlib-source]/configure --target=arm-elf --prefix=[toolchain-prefix] --enable-
interwork--enable-multilib
(10)make all install
(11)cd [gcc-build]
(12)make all install
(13)cd [gdb-build]
(14)[gdb-source]/configure --target=arm-elf --prefix=[toolchain-prefix] --enable-inter
work --enable-multilib
(15)make all install
```

当使用交叉编译器的时候，程序通常用前缀来指示目标的体系结构和连接器的输出格式，如 arm-linux-gcc、arm-linux-gdb、powerpc-linux-gcc 等。

建立交叉工具链的过程相当繁琐，我们不必亲自操作，可以直接下载第三方编译好了的、开放的、针对目标处理器的交叉工具链，如在 http://www.codesourcery.com/gnu_toolchains/上可以下载针对 ARM、ColdFire 和 Power 的工具链。

本书配套光盘提供的 VirtualBox 虚拟机映像中，已经包含了制作好的 arm-linux-gcc 等。

22.1.3 串口工具

在嵌入式 Linux 的调试过程中，目标机往往会提供给主机一个串口控制台，驱动工程师在 80%以上的情况下都是通过串口与目标机通信。因此，好用的串口工具将大大提高工程师的生产效率。

在 Windows 环境下，其附件内自带了超级终端，超级终端包括了对 VT100、ANSI 等终端仿真功能以及对 xmodem、ymodem、zmodem 等协议的支持，如图 22.4 所示。

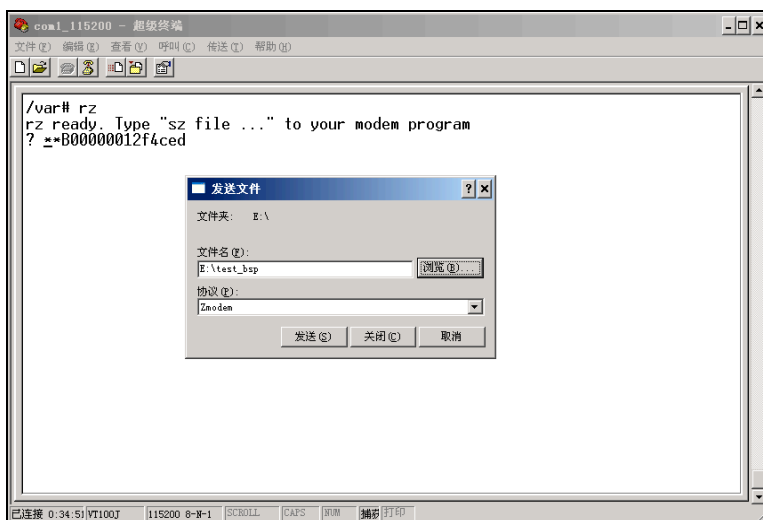


图 22.4 超级终端

在调试过程中，经常需要保存串口打印信息的历史记录，这时候可以使用“传送”菜单下的“捕获文字”功能来实现。

SecureCRT 是比超级终端更强大且更方便的工具，它将 SSH 的安全登录、数据传送性能和 Windows 终端仿真提供的可靠性、可用性和可配置性结合在一起，其界面如图 22.5 所示。鉴于 SecureCRT 具备比超级终端更强大且好用的功能，建议直接用 SecureCRT 替代超级终端。

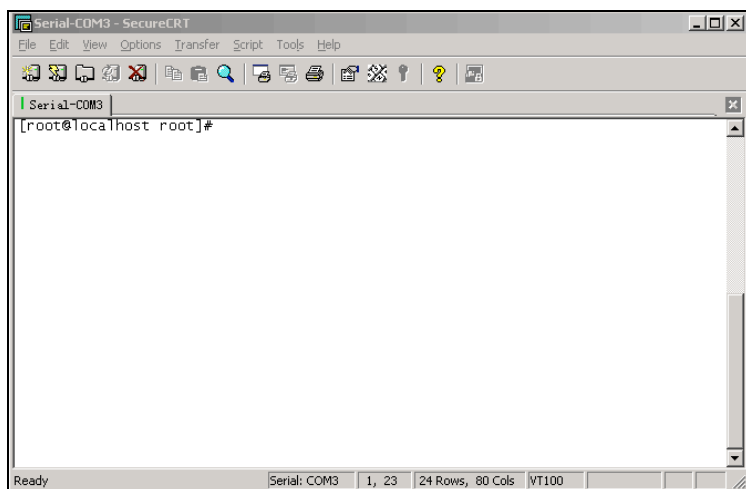


图 22.5 SecureCRT

在开发过程中，为执行自动化的串口发送操作，可以使用 SecureCRT 的 VBScript 脚本功能，让其运行一段脚本，自动捕获接收到的串口信息并向串口上发送指定的数据或文件。代码清单 22.1 所示的脚本等待接收到“CCC”字符串后通过 xmodem 协议发送 file.bin 文件，之后，当接收到“y/n”时，选择“y”。

代码清单 22.1 SecureCRT VBScript 脚本范例

```

1 #Language = "VBScript"
2 #Interface = "1.0"
3
4 Sub main
5   Dir = "d:\baohua\"
6   ' turn on synchronous mode so we don't miss any data
7   crt.Screen.Synchronous = True
8   'wait "CCC" string then send file
9   crt.Screen.WaitForString "CCC"
10  crt.FileTransfer.SendXmodem Dir & "file.bin"
11  'wait "y/n" string then send "y"
12  crt.Screen.WaitForString "y/n"
13  crt.Screen.Send "y" & VbCr
14 End Sub

```

另外，在 Windows 下环境下，也可以选用 PuTTY 工具，该工具非常小巧，功能很强大，支持串口、Telnet 和 SSH 等，其官方网址为：<http://www.chiark.greenend.org.uk/~sgtatham/putty/>。

Minicom 是 Linux 系统下常用的类似于 Windows 下超级终端的工具，当要发送文件或设置串口时，需先按下“CTRL+A”，紧接着按下“Z”键激活菜单，如图 22.6 所示。



图 22.6 Minicom

除了 Minicom 以外，在 Linux 系统下，也可以直接使用 C-Kermit。运行 kermit 命令即可启动 C-Kermit。在使用 C-Kermit 连接目标板之前，需先进行串口设置，如下所示：

```

set line /dev/ttyS0
set speed 115200
set carrier-watch off
set handshake none
set flow-control none
robust
set file type bin
set file name lit
set rec pack 1000
set send pack 1000
set window 5

```



之后, 使用以下命令就可以连接到目标板:

```
connect
```

在 `kermit` 的使用过程中, 会涉及串口控制台和 `kermit` 功能模式之间的切换, 从串口控制台切换到 `kermit` 的方法是按下 “`Ctrl+\`”, 然后再按下 “`C`”。

假设我们在串口控制台上敲入命令使得目标板进入文件接收等待状态, 此后可按下 “`Ctrl+\`”, 再按 “`C`”, 切换到 `kermit`, 运行 “`send /file_name`” 命令传输文件。文件传输结束后, 再运行 “`c`” 命令将进入串口控制台。

22.2 GDB 调试器用法

22.2.1 GDB 基本用法

GDB 是 GNU 开源组织发布的一个强大的 UNIX 下的程序调试工具, GDB 主要可帮助工程师完成下面 4 个方面的功能。

- 启动程序, 可以按照工程师自定义的要求运行程序。
- 让被调试的程序在工程师指定的断点处停住, 断点可以是条件表达式。
- 当程序被停住时, 可以检查此时程序中所发生的事, 并追踪上文。
- 动态地改变程序的执行环境。

不管是调试 Linux 内核空间的驱动还是调试用户空间的应用程序, 掌握 GDB 的用法都是必须。而且, 调试内核和调试应用程序时使用的 GDB 命令是完全相同的, 下面以代码清单 22.2 的应用程序为例演示 GDB 调试器的用法。

代码清单 22.2 GDB 调试演示程序

```
1 int add(int a, int b)
2 {
3     return a + b;
4 }
5
6 main()
7 {
8     int sum[10] =
9     {
10         0, 0, 0, 0, 0, 0, 0, 0, 0, 0
11     };
12     int i;
13
14     int array1[10] =
15     {
16         48, 56, 77, 33, 33, 11, 226, 544, 78, 90
17     };
18     int array2[10] =
19     {
20         85, 99, 66, 0x199, 393, 11, 1, 2, 3, 4
21     };
```

```

22
23   for (i = 0; i < 10; i++)
24   {
25       sum[i] = add(array1[i], array2[i]);
26   }
27 }

```

使用命令 “`gcc -g gdb_example.c -o gdb_example`” 编译上述程序，得到包含调试信息的二进制文件 `example`，执行 “`gdb gdb_example`” 命令进入调试状态，如下所示：

```

[root@localhost driver_study]# gdb gdb_example
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb)

```

1. list 命令

在 GDB 中运行 `list` 命令（缩写 `l`）可以列出代码，`list` 的具体形式如下。

- `list <linenum>`，显示程序第 `linenum` 行周围的源程序，如下所示：

```

(gdb) list 15
10
11     int array1[10] =
12     {
13         48, 56, 77, 33, 33, 11, 226, 544, 78, 90
14     };
15     int array2[10] =
16     {
17         85, 99, 66, 0x199, 393, 11, 1, 2, 3, 4
18     };
19

```

- `list <function>`，显示函数名为 `function` 的函数的源程序，如下所示：

```

(gdb) list main
2     {
3         return a + b;
4     }
5
6     main()
7     {
8         int sum[10];
9         int i;
10
11     int array1[10] =

```

- `list`，显示当前行后面的源程序。
- `list -`，显示当前行前面的源程序。

下面演示了使用 GDB 中的 `run`（缩写为 `r`）、`break`（缩写为 `b`）、`next`（缩写为 `n`）命令控制程序的运行，并使用 `print`（缩写为 `p`）命令打印程序中的变量 `sum` 的过程：

```

(gdb) break add
Breakpoint 1 at 0x80482f7: file gdb_example.c, line 3.
(gdb) run

```



```
Starting program: /driver_study/gdb_example

Breakpoint 1, add (a=48, b=85) at gdb_example.c:3
warning: Source file is more recent than executable.

3      return a + b;
(gdb) next
4      }
(gdb) next
main () at gdb_example.c:23
23      for (i = 0; i < 10; i++)
(gdb) next
25      sum[i] = add(array1[i], array2[i]);
(gdb) print sum
$1 = {133, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

2. run 命令

在 GDB 中，运行程序使用 `run` 命令。在程序运行前，我们可以设置如下 4 方面的工作环境。

(1) 程序运行参数。

“`set args`”可指定运行时参数，如：“`set args 10 20 30 40 50`”；“`show args`”命令可以查看设置好的运行参数。

(2) 运行环境。

“`path <dir>`”可设定程序的运行路径；“`show paths`”可查看程序的运行路径；“`set environment varname [=value]`”用于设置环境变量，如“`set env USER=baohua`”；“`show environment [varname]`”则用于查看环境变量。

(3) 工作目录。

“`cd <dir>`”相当于 shell 的 `cd` 命令，`pwd` 显示当前所在的目录。

(4) 程序的输入输出。

“`info terminal`”用于显示程序用到的终端的模式；GDB 中也可以使用重定向控制程序输出，如“`run > outfile`”；`tty` 命令可以指定输入输出的终端设备，如：`tty /dev/ttyS1`。

3. break 命令

在 GDB 中用 `break` 命令来设置断点，设置断点的方法如下。

(1) `break <function>`。

在进入指定函数时停住，C++中可以使用“`class::function`”或“`function(type, type)`”格式来指定函数名。

(2) `break <linenum>`。

在指定行号停住。

(3) `break +offset / break - offset`。

在当前行号的前面或后面的 `offset` 行停住，`offset` 为自然数。

(4) `break filename:linenum`。

在源文件 `filename` 的 `linenum` 行处停住。

(5) `break filename:function`。

在源文件 `filename` 的 `function` 函数的入口处停住。

(6) `break *address`。

在程序运行的内存地址处停住。

(7) `break`。

`break` 命令没有参数时，表示在下一条指令处停住。

(8) `break...if <condition>`。

“...”可以是上述的“`break <linenum>`”、“`break +offset / break -offset`”中的参数，`condition` 表示条件，在条件成立时停住。比如在循环体中，可以设置“`break if i=100`”，表示当 `i` 为 100 时停住程序。

查看断点时，可使用 `info` 命令，如“`info breakpoints [n]`”、“`info break [n]`”（`n` 表示断点号）。

4. 单步命令

在调试过程中，`next` 命令用于单步执行，类似于 VC++ 中的 `step over`。`next` 的单步不会进入函数的内部，与 `next` 对应的 `step`（缩写为 `s`）命令则在单步执行一个函数时，会进入其内部，类似于 VC++ 中的 `step into`。下面演示了 `step` 命令的执行情况，在第 23 行的 `add()` 函数调用处执行 `step` 会进入其内部的“`return a+b;`”语句：

```
(gdb) break 25
Breakpoint 1 at 0x8048362: file gdb_example.c, line 25.
(gdb) run
Starting program: /driver_study/gdb_example

Breakpoint 1, main () at gdb_example.c:25
25      sum[i] = add(array1[i], array2[i]);
(gdb) step
add (a=48, b=85) at gdb_example.c:3
3      return a + b;
```

单步执行的更复杂用法如下。

(1) `step <count>`。

单步跟踪，如果有函数调用，则进入该函数（进入函数的前提是，此函数被编译有 `debug` 信息）。`step` 后面不加 `count` 表示一条条地执行，加 `count` 表示执行后面的 `count` 条指令，然后再停住。

(2) `next <count>`。

单步跟踪，如果有函数调用，它不会进入该函数。同样地，`next` 后面不加 `count` 表示一条条地执行，加 `count` 表示执行后面的 `count` 条指令，然后再停住。

(3) `set step-mode`。

“`set step-mode on`”用于打开 `step-mode` 模式，这样，在进行单步跟踪时，程序不会因为没有 `debug` 信息而不停住，这个参数的设置可便于查看机器码。“`set step-mode off`”用于关闭 `step-mode` 模式。

(4) `finish`。

运行程序，直到当前函数完成返回，并打印函数返回时的堆栈地址和返回值及参数值等信息。

(5) `until`（缩写为 `u`）。

一直在循环体内执行单步，退不出来是一件令人烦恼的事情，`until` 命令可以运行程序直到退出循环体。



(6) stepi (缩写为 si) 和 nexti (缩写为 ni)。

stepi 和 nexti 用于单步跟踪一条机器指令，一条程序代码有可能由数条机器指令完成，stepi 和 nexti 可以单步执行机器指令。

另外，运行 “display/i \$pc” 命令后，单步跟踪会在打出程序代码的同时打出机器指令，即汇编代码。

5. continue 命令

当程序被停住后，可以使用 continue 命令 (缩写为 c, fg 命令同 continue 命令) 恢复程序的运行直到程序结束，或到达下一个断点，命令格式为：

```
continue [ignore-count]
c [ignore-count]
fg [ignore-count]
```

ignore-count 表示忽略其后多少次断点。

假设我们设置了函数断点 add(), 并 watch i, 则在 continue 过程中，每次遇到 add() 函数或 i 发生变化，程序就会停住，如下所示：

```
(gdb) continue
Continuing.
Hardware watchpoint 3: i

Old value = 2
New value = 3
0x0804838d in main () at gdb_example.c:23
23      for (i = 0; i < 10; i++)
(gdb) continue
Continuing.

Breakpoint 1, main () at gdb_example.c:25
25      sum[i] = add(array1[i], array2[i]);
(gdb) continue
Continuing.
Hardware watchpoint 3: i

Old value = 3
New value = 4
0x0804838d in main () at gdb_example.c:23
23      for (i = 0; i < 10; i++)
```

6. print 命令

在调试程序时，当程序被停住时，可以使用 print 命令 (缩写为 p)，或是同义命令 inspect 来查看当前程序的运行数据。print 命令的格式如下：

```
print <expr>
print /<f> <expr>
```

<expr> 是表达式，是被调试的程序中的表达式，<f> 是输出的格式，比如，如果要把表达式按十六进制的格式输出，那么就是/x。在表达式中，有几种 GDB 所支持的操作符，它们可以用在任何一种语言中，“@” 是一个和数组有关的操作符，“::” 指定一个在文件或是函数中的变量，“{<type>} <addr>” 表示一个指向内存地址 <addr> 的类型为 type 的一个对象。

下面演示了查看 sum[] 数组的值的過程：

```
(gdb) print sum
$2 = {133, 155, 0, 0, 0, 0, 0, 0, 0, 0}
(gdb) next

Breakpoint 1, main () at gdb_example.c:25
25      sum[i] = add(array1[i], array2[i]);
(gdb) next
23      for (i = 0; i < 10; i++)
(gdb) print sum
$3 = {133, 155, 143, 0, 0, 0, 0, 0, 0, 0}
```

当需要查看一段连续内存空间的值的时间，可以使用 GDB 的 “@” 操作符，“@” 的左边是第一个内存地址，“@” 的右边则是想查看内存的长度。例如如下动态申请的内存：

```
int *array = (int *) malloc (len * sizeof (int));
```

在 GDB 调试过程中这样显示出这个动态数组的值：

```
p *array@len
```

print 的输出格式如下。

- x: 按十六进制格式显示变量。
- d: 按十进制格式显示变量。
- u: 按十六进制格式显示无符号整型。
- o: 按八进制格式显示变量。
- t: 按二进制格式显示变量。
- a: 按十六进制格式显示变量。
- c: 按字符格式显示变量。
- f: 按浮点数格式显示变量。

我们可用 display 命令设置一些自动显示的变量，当程序停住时，或是单步跟踪时，这些变量会自动显示。

如果要修改变量，如 x 的值，可使用如下命令：

```
print x=4
```

当用 GDB 的 print 查看程序运行时的数据时，每一个 print 都会被 GDB 记录下来。GDB 会以 \$1, \$2, \$3 … 这样的方式为每一个 print 命令编号。我们可以使用这个编号访问以前的表达式，如 \$1。

7. watch 命令

watch 一般来观察某个表达式（变量也是一种表达式）的值是否有变化了，如果有变化，马上停止程序运行。我们有如下几种方法来设置观察点。

watch <expr>: 为表达式（变量）expr 设置一个观察点。一旦表达式值有变化时，马上停止程序运行。

rwatch <expr>: 当表达式（变量）expr 被读时，停止程序运行。

awatch <expr>: 当表达式（变量）的值被读或被写时，停止程序运行。

info watchpoints: 列出当前所设置了的所有观察点。

下面演示了观察 i 并在连续运行 next 时一旦发现 i 变化，i 值就会显示出来的过程：

```
(gdb) watch i
Hardware watchpoint 3: i
(gdb) next
```




```
23      for (i = 0; i < 10; i++)
(gdb) next
Hardware watchpoint 3: i

Old value = 0
New value = 1
0x0804838d in main () at gdb_example.c:23
23      for (i = 0; i < 10; i++)
(gdb) next

Breakpoint 1, main () at gdb_example.c:25
25      sum[i] = add(array1[i], array2[i]);
(gdb) next
23      for (i = 0; i < 10; i++)
(gdb) next
Hardware watchpoint 3: i

Old value = 1
New value = 2
0x0804838d in main () at gdb_example.c:23
23      for (i = 0; i < 10; i++)
```

8. examine 命令

我们可以使用 `examine` 命令（缩写为 `x`）来查看内存地址中的值。`examine` 命令的语法如下所示：

```
x/<n/f/u> <addr>
```

`<addr>` 表示一个内存地址。“`x`”后的 `n`、`f`、`u` 都是可选的参数，`n` 是一个正整数，表示显示内存的长度，也就是说从当前地址向后显示几个地址的内容；`f` 表示显示的格式，如果地址所指的是字符串，那么格式可以是 `s`，如果地址是指令地址，那么格式可以是 `i`；`u` 表示从当前地址往后请求的字节数，如果不指定的话，GDB 默认是 4 字节。`u` 参数可以被一些字符代替：`b` 表示单字节，`h` 表示双字节，`w` 表示四字节，`g` 表示八字节。当我们指定了字节长度后，GDB 会从指定的内存地址开始，读写指定字节，并把其当作一个值取出来。`n`、`f`、`u` 这 3 个参数可以一起使用，例如命令 “`x/3uh 0x54320`” 表示从内存地址 `0x54320` 开始以双字节为 1 个单位（`h`）、16 进制方式（`u`）显示 3 个单位（3）的内存。

9. jump 命令

一般来说，被调试程序会按照程序代码的运行顺序依次执行，但是 GDB 也提供了乱序执行的功能，也就是说，GDB 可以修改程序的执行顺序，从而让程序随意跳跃。这个功能可以由 GDB 的 `jump` 命令 “`jump <linespec>`” 来指定下一条语句的运行点。`<linespec>` 可以是文件的行号，可以是 “`file:line`” 格式，也可以是 “`+num`” 这种偏移量格式，表示下一条运行语句从哪里开始。

```
jump <address>
```

这里的 `<address>` 是代码行的内存地址。

注意，`jump` 命令不会改变当前的程序栈中的内容，所以，如果使用 `jump` 从一个函数跳转到另一个函数，当跳转到的函数运行完返回，进行出栈操作时必然会发生错误，这可能导致意想不到的结果，所以最好只用 `jump` 在同一个函数中进行跳转。

10. signal 命令

使用 `signal` 命令，可以产生一个信号量给被调试的程序，如中断信号 “`Ctrl+C`”。这非常方便

于程序的调试，可以在程序运行的任意位置设置断点，并在该断点用 GDB 产生一个信号量，这种精确地在某处产生信号的方法非常有利于程序的调试。

signal 命令的语法是“signal <signal>”，UNIX 的系统信号量通常为 1~15，所以<signal>取值也在这个范围。

11. return 命令

如果在函数中设置了调试断点，在断点后还有语句没有执行完，这时候我们可以使用 return 命令强制函数忽略还没有执行的语句并返回。

```
return
return <expression>
```

上述 return 命令用于取消当前函数的执行，并立即返回，如果指定了<expression>，那么该表达式的值会被作为函数的返回值。

12. call 命令

call 命令用于强制调用某函数：

```
call <expr>
```

表达式可以是函数，以此达到强制调用函数的目的，它会显示函数的返回值（如果函数返回值不是 void）。

其实，前面介绍的 print 命令也可以完成强制调用函数的功能。

13. info 命令

info 命令可以在调试时用来查看寄存器、断点、观察点和信号等信息。要查看寄存器的值，可以使用如下命令：

```
info registers （查看除了浮点寄存器以外的寄存器）
info all-registers （查看所有寄存器，包括浮点寄存器）
info registers <regname ...> （查看所指定的寄存器）
```

要查看断点信息，可以使用如下命令：

```
info break
```

列出当前所设置的所有观察点，使用如下命令：

```
info watchpoints
```

查看有哪些信号正在被 gdb 检测，使用如下命令：

```
info signals
info handle
```

也可以使用 info line 命令来查看源代码在内存中的地址。info line 后面可以跟行号、函数名、文件名：行号、文件名:函数名等多种形式，例如下面的命令会打印出所指定的源码在运行时的内存地址：

```
info line tst.c:func
```

14. disassemble

disassemble 命令用于反汇编，它可被用来查看当前执行时的源代码的机器码，实际上只是把目前内存中的指令 dump 出来。下面的示例用于查看函数 func 的汇编代码：

```
(gdb) disassemble func
Dump of assembler code for function func:
0x8048450 <func>:      push    %ebp
0x8048451 <func+1>:    mov     %esp,%ebp
0x8048453 <func+3>:    sub     $0x18,%esp
```



```
0x8048456 <func+6>:    movl    $0x0,0xffffffffc(%ebp)
...
End of assembler dump.
```

22.2.2 DDD 图形界面调试工具

GDB 本身是一种命令行调试工具，但是通过 DDD (Data Display Debugger, <http://www.gnu.org/software/ddd/>), 可以被图形界面化。DDD 可以作为 GDB、DBX、WDB、Ladebug、JDB、XDB、Perl Debugger 或 Python Debugger 的可视化图形前端, 其特有的图形数据显示功能 (Graphical Data Display) 可以把数据结构按照图形的方式显示出来。

DDD 最初源于 1990 年 Andreas Zeller 编写的 VSL 结构化语言, 后来经过一些程序员的努力, 演化成今天的模样。DDD 的功能非常强大, 可以调试用 C/C++、Ada、Fortran、Pascal、Modula-2 和 Modula-3 编写的程序; 可以超文本方式浏览源代码; 能够进行断点设置、回溯调试和历史记录编辑; 具有程序在终端运行的仿真窗口, 具备在远程主机上进行调试的能力; 能够显示各种数据结构之间的关系, 并将数据结构以图形化形式显示; 具有 GDB/DBX/XDB 的命令行界面, 包括完全的文本编辑、历史纪录、搜寻引擎等。

DDD 的主界面如图 22.7 所示, 和 Visual Studio 等集成开发环境非常相近, 而且 DDD 包含了 Visual Studio 所不包含的部分功能。

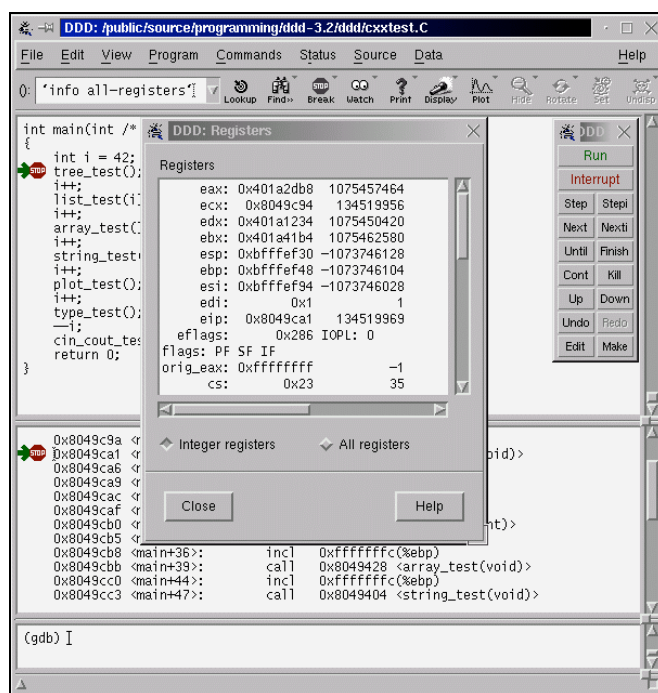


图 22.7 DDD 的主界面

在设计 DDD 的时候, 设计人员决定把它与 GDB 之间的耦合度尽可能降低。因为像 GDB 这样的开源软件, 更新比商业软件快。所以为了使 GDB 的变化不会影响到 DDD, 在 DDD 中, GDB 是作为独立的进程运行的, 通过命令行接口与 DDD 进行交互。

图 22.8 显示了用户、DDD、GDB 和被调试进程之间的关系，DDD 和 GDB 之间的所有通信都是异步进行的。在 DDD 中发出的 GDB 命令都会与一个回调例程相连，放入命令队列中。这个回调例程在合适的时间会处理 GDB 的输出。例如，如果用户手动输入一条 GDB 的命令，DDD 就会把这条命令与显示 GDB 输出的一个回调例程连起来。一旦 GDB 命令完成，就会触发回调例程，GDB 的输出就会显示在 DDD 的命令窗口中。

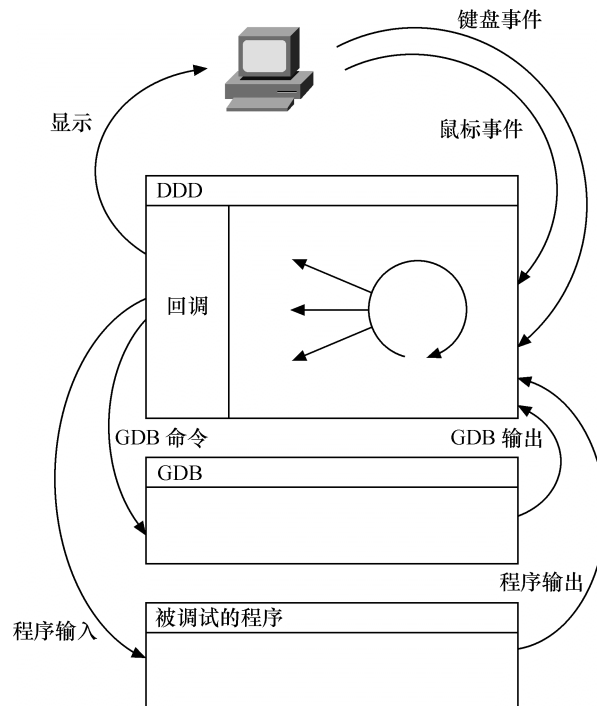


图 22.8 DDD 运行机理

DDD 在事件循环时等待用户输入和 GDB 输出，同时等着 GDB 进入等待输入状态。当 GDB 可用时，下一条命令就会从命令队列中取出，送给 GDB。GDB 到达的输出由上次命令的回调过程来处理。这种异步机制避免了 DDD 在等待 GDB 输出时发生阻塞现象，到达的事件可以在任何时间得到处理。

不可否认的是，DDD 和 GDB 的分离使得 DDD 运行速度相对来说比较慢，但是这种方法带来了灵活性和兼容性的好处。例如，用户可以把 GDB 调试器换成其他调试器，如 DBX 等。另外，GDB 和 DDD 的分离使得用户可以在不同的机器上分别运行 GDB 和 DDD。

在 DDD 中，可以直接在底部的控制台中输入 GDB 命令，也可以通过菜单和鼠标来以图形方式触发 GDB 命令的运行，使用方法甚为简单，因此这里不再赘述。除了基本的 GDB 命令外，DDD 中的 Plot 工具可以用于将数组以二维或三维坐标系中点、曲线或曲面的方式显示出来，如图 22.9 所示，这在某些场合下会非常有用。dsp 工程师应该不会陌生，因为在 dsp 程序调试中，在集成开发环境中绘制数组曲线是十分常见的用法。

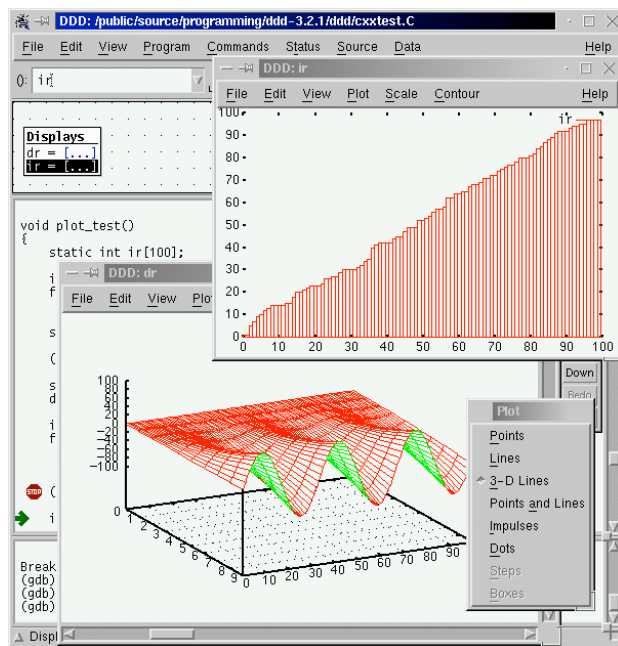


图 22.9 DDD 中的 Plot 绘制数组

DDD 不仅可用于调试 PC 上的应用程序，也可调试目标板子，方法是用如下命令启动 DDD：

```
ddd -debugger arm-linux-gdb <要调试的程序>
```

22.3 Linux 内核调试

在嵌入式系统中，由于目标机资源有限，因此往往在主机上编译好程序，再在目标机上运行。用户所有的开发工作在主机开发环境下完成，包括编码、编译、连接、下载和调试等。目标机和主机通过串口、以太网、仿真器或其他通信手段通信，主机用这些接口控制目标机，调试目标机上的程序。

调试嵌入式 Linux 内核的方法如下。

(1) 目标机“插桩”，如打上 KGDB 补丁，这样主机上的 GDB 可与目标机的 KGDB 通过串口或网口通信。

(2) 使用仿真器，仿真器可直接连接目标机的 JTAG/BDM，这样主机的 GDB 就可以通过与仿真器的通信来控制目标机。

(3) 在目标板上通过 `printk()`、`oops`、`strace` 等软件方法进行“观察”调试，这些方法不具备查看和修改数据结构、断点、单步等功能。

第 22.3~22.7 节将对这些调试方法进行一一讲解。

不管是目标机“插桩”还是使用仿真器连接目标机 JTAG/BDM，在主机上，调试工具一般都采用 GDB。尽管采用“插桩”和仿真器的方式可以进行查看和修改数据结构、断点、单步等，而 `printk()` 这种最原始的方法却更广泛地被应用。

22.4 内核打印信息——printk()

在 Linux 中，内核打印语句 `printk()` 会将内核信息输出到内核信息缓冲区中。内核信息缓冲区是一个环形缓冲区（ring buffer），因此，如果塞入的消息过多，就会将之前的消息冲刷掉。

Linux 的 `klogd` 进程（一个系统守护进程，它截获并且记录下 Linux 内核日志信息）会通过 `/proc/kmsg` 文件读取缓冲区，一旦读取完成，内核信息便从缓冲区中被删除。之后，`klogd` 守护进程会将读取的内核信息派发给 `syslogd` 守护进程（`syslogd` 记录下系统里所有提供日志记录的程序给出的日志和信息内容，每一个被记录的消息至少包含时间戳和主机名），`syslogd` 这个守护进程会根据 `/etc/syslog.conf` 将不同的服务产生的日志记录到不同的文件中。例如在 `/etc/syslog.conf` 中增加 “`kern.* /tmp/kernel_debug.txt`” 一行，则内核信息也会被放置到 `/tmp/kernel_debug.txt` 文件中。执行 “`insmod hello.ko`”，我们看到 `/tmp/kernel_debug.txt` 文件中多出如下一行：

```
Jul 6 00:40:51 localhost kernel: Hello World enter
```

用户也可以直接使用 “`cat /proc/kmsg`” 命令来显示内核信息，但是，由于 `/proc/kmsg` 是一个“永无休止的文件”，因此，“`cat /proc/kmsg`” 的进程只能通过 “`Ctrl+C`” 或 `kill` 终止。另外，使用 `dmesg` 命令也可以直接读取 ring buffer 中的信息。

`printk()` 定义了 8 个消息级别，分为级别 0~7，越低级别（数值越大）的消息越不重要，第 0 级是紧急事件级，第 7 级是调试级，代码清单 22.3 所示为 `printk()` 的级别定义。

代码清单 22.3 `printk()` 的级别定义

```
1 #define KERN_EMERG "<0>" /*紧急事件，一般是系统崩溃之前提示的消息 */
2 #define KERN_ALERT "<1>" /* 必须立即采取行动 */
3 #define KERN_CRIT "<2>" /*临界状态，通常涉及严重的硬件或软件操作失败 */
4 #define KERN_ERR "<3>" /*用于报告错误状态，设备驱动程序会
5 经常使用 KERN_ERR 来报告来自硬件的问题 */
6 #define KERN_WARNING "<4>" /*对可能出现问题的情况进行警告，
7 这类情况通常不会对系统造成严重问题 */
8 #define KERN_NOTICE "<5>" /*有必要进行提示的正常情形，
9 许多与安全相关的状况用这个级别进行汇报*/
10 #define KERN_INFO "<6>" /*内核提示性信息，很多驱动程序
11 在启动的时候，以这个级别打印出它们找到的硬件信息*/
12 #define KERN_DEBUG "<7>" /* 用于调试信息 */
```

通过 `/proc/sys/kernel/printk` 文件可以调节 `printk` 的输出等级，该文件有 4 个数字值，如下所示。

- 控制台日志级别：优先级高于该值的消息将被打印至控制台。
- 默认的消息日志级别：将用该优先级来打印没有优先级的消息。
- 最低的控制台日志级别：控制台日志级别可被设置的最小值（最高优先级）。
- 默认的控制台日志级别：控制台日志级别的默认值。

上述 4 个值的默认设置为 6、4、1、7。

通过如下命令可以使得 Linux 内核的任何 `printk` 都被输出：

```
# echo 8 > /proc/sys/kernel/printk
```

在设备驱动中，我们经常需要输出调试或系统信息，尽管可以直接采用 `printk("<7>debug info ...\\n")` 方式的 `printk()` 语句输出，但是通常可以使用封装了 `printk()` 的更高级的宏，如



pr_debug()、dev_debug()等。代码清单 22.4 所示为 pr_debug()和 pr_info()的定义,代码清单 22.5 所示为 dev_dbg()、dev_err()、dev_info()等的定义,前一组的输出中不包含设备信息,后一组包含。

代码清单 22.4 替代 printk()的宏

```
1 #ifdef DEBUG
2 #define pr_debug(fmt,arg...) \
3     printk(KERN_DEBUG fmt,##arg)
4 #else
5 static inline int __attribute__((format (printf, 1, 2))) pr_debug(const char * fmt, ...)
6 {
7     return 0;
8 }
9 #endif
10
11 #define pr_info(fmt,arg...) \
12     printk(KERN_INFO fmt,##arg)
```

代码清单 22.5 包含设备信息的替代 printk()的宏

```
1 #define dev_printk(level, dev, format, arg...) \
2     printk(level "%s %s: " format , dev_driver_string(dev) , (dev)->bus_id , ## arg)
3
4 #ifdef DEBUG
5 #define dev_dbg(dev, format, arg...) \
6     dev_printk(KERN_DEBUG , dev , format , ## arg)
7 #else
8 #define dev_dbg(dev, format, arg...) do { (void)(dev); } while (0)
9 #endif
10
11 #define dev_err(dev, format, arg...) \
12     dev_printk(KERN_ERR , dev , format , ## arg)
13 #define dev_info(dev, format, arg...) \
14     dev_printk(KERN_INFO , dev , format , ## arg)
15 #define dev_warn(dev, format, arg...) \
16     dev_printk(KERN_WARNING , dev , format , ## arg)
17 #define dev_notice(dev, format, arg...) \
18     dev_printk(KERN_NOTICE , dev , format , ## arg)
```

在打印信息时,如果想输出其所在的函数,可以使用 __FUNCTION__, 如:

```
printk("%s: Incorrect IRQ %d from %s\n", __FUNCTION__, irq, devname);
```

C99 标准已经提供了 __func__ 来指定函数名,因此目前 __FUNCTION__ 实际定义为:

```
#define __FUNCTION__ (__func__)
```

22.5 使用 /proc

在 Linux 系统中, /proc 文件系统十分有用,它被用于内核向用户导出信息。/proc 文件系统是一个虚拟文件系统,通过它可以使用一种新的方法在 Linux 内核空间 and 用户空间之间进行通信。在 /proc 文件系统中,我们可以将对虚拟文件的读写作为与内核中实体进行通信的一种手段,与普通文件不同的是,这些虚拟文件的内容都是动态创建的。/proc 下的文件并非完全是只读的,若节

点可写，还可用于一定的控制或配置目的，例如前面介绍的写 `/proc/sys/kernel/printk` 可以改变 `printk` 的打印级别。

Linux 系统的许多命令本身都是通过分析 `/proc` 下的文件来完成，如 `ps`、`top`、`uptime` 和 `free` 等。例如，`free` 命令通过分析 `/proc/meminfo` 文件得到可用内存信息，下面显示了对应的 `meminfo` 文件和 `free` 命令的结果。

- `meminfo` 文件：

```
[root@localhost proc]# cat meminfo
MemTotal:      29516 kB
MemFree:       1472 kB
Buffers:       4096 kB
Cached:       12648 kB
SwapCached:    0 kB
Active:       14208 kB
Inactive:     8844 kB
HighTotal:    0 kB
HighFree:    0 kB
LowTotal:    29516 kB
LowFree:    1472 kB
SwapTotal:   265064 kB
SwapFree:   265064 kB
Dirty:       20 kB
Writeback:    0 kB
Mapped:     10052 kB
Slab:       3864 kB
CommitLimit: 279820 kB
Committed_AS: 13760 kB
PageTables:   444 kB
VmallocTotal: 999416 kB
VmallocUsed:   560 kB
VmallocChunk: 998580 kB
```

- `free` 命令：

```
[root@localhost proc]# free
              total        used        free      shared    buffers     cached
Mem:      29516      28104      1412         0        4100       12700
-/+ buffers/cache: 11304      18212
Swap:    265064         0      265064
```

Linux 的 USB、PCI 等内核代码本身都会创建 `/proc` 节点导出内核信息，虽然不值得鼓励，但在 Linux 设备驱动程序中，驱动工程师自定义 `/proc` 节点以向外界传递信息的方法仍然是可行的。

在 Linux 系统中，可用如下函数创建 `/proc` 节点：

```
struct proc_dir_entry *create_proc_entry(const char *name, mode_t mode,
                                         struct proc_dir_entry *parent);

struct proc_dir_entry *create_proc_read_entry(const char *name, mode_t mode,
                                              struct proc_dir_entry *base, read_proc_t *read_proc, void * data);
```

`create_proc_entry()` 函数用于创建 `/proc` 节点，而 `create_proc_read_entry()` 调用 `create_proc_entry()` 创建只读的 `/proc` 节点。参数 `name` 为 `/proc` 节点的名称，`parent/base` 为父目录的节点，如果为 `NULL`，则指 `/proc` 目录，`read_proc` 是 `/proc` 节点的读函数指针。当 `read()` 系统调用在 `/proc` 文件系统中执行时，它映像到一个数据产生函数，而不是一个数据获取函数。



下列函数用于创建/proc 目录:

```
struct proc_dir_entry *proc_mkdir(const char *name, struct proc_dir_entry *parent);
```

结合 create_proc_entry()和 proc_mkdir(), 代码清单 22.6 中的程序可用于先在/proc 下创建一个目录, 而后在该目录下创建一个文件。

代码清单 22.6 proc_mkdir()和 create_proc_entry()函数使用范例

```
1 /* 创建/proc 下的目录 */
2 example_dir = proc_mkdir("procfs_example", NULL);
3 if (example_dir == NULL) {
4     rv = - ENOMEM;
5     goto out;
6 }
7
8 example_dir->owner = THIS_MODULE;
9
10 /* 创建一个例子/proc 文件 */
11 example_file = create_proc_entry("example_file", 0666, example_dir);
12 if (example_file == NULL) {
13     rv = - ENOMEM;
14     goto out;
15 }
16
17 example_file->owner = THIS_MODULE;
18 example_file->read_proc = example_file_read;
19 example_file->write_proc = example_file_write;
```

作为上述函数各返回值的 proc_dir_entry 结构体中包含了/proc 节点的读函数指针(read_proc_t *read_proc)、写函数指针 (write_proc_t *write_proc) 以及父节点、子节点信息等。

/proc 节点的读写函数的类型分别为:

```
typedef int (read_proc_t)(char *page, char **start, off_t off,
                          int count, int *eof, void *data);
typedef int (write_proc_t)(struct file *file, const char __user *buffer,
                           unsigned long count, void *data);
```

读函数中 page 指针指向用于写入数据的缓冲区, start 用于返回实际的数据写到内存页的位置, eof 是用于返回读结束标志, offset 是读的偏移, count 是要读的数据长度。

start 参数比较复杂, 对于/proc 只包含简单数据的情况, 通常不需要在读函数中设置*start, 意味着内核将认为数据保存在内存页偏移 0 的地方。如果将*start 设置为非 0 值, 意味着内核将认为*start 指向的数据是 offset 偏移处的数据。

写函数与 file_operations 中的 write()成员类似, 需要一次从用户缓冲区到内存空间的复制过程。

Linux 系统中可用如下函数删除/proc 节点:

```
void remove_proc_entry(const char *name, struct proc_dir_entry *parent);
```

Linux 系统中已经定义好的可使用的/proc 节点宏包括: proc_root_fs (/proc)、proc_net (/proc/net)、proc_bus (/proc/bus)、proc_root_driver (/proc/driver) 等, proc_root_fs 实际就是 NULL。

代码清单 22.7 所示为一个简单的/proc 使用范例, 这段代码在模块加载函数中创建/proc 文件节点, 在模块卸载函数中撤销/proc 节点, 而文件中只保存了一个 32 位的无符号整形值。

代码清单 22.7 /proc 文件系统使用模板

```

1 #include ...
2
3 static struct proc_dir_entry *proc_entry;
4 static unsigned long val = 0x12345678;
5
6 /* 读/proc 文件接口 */
7 ssize_t simple_proc_read(char *page, char **start, off_t off, int count,
8   int*eof, void *data)
9 {
10   int len;
11   if (off > 0) { /* 不能偏移访问 */
12     *eof = 1;
13     return 0;
14   }
15
16   len = sprintf(page, "%08x\n", val);
17
18   return len;
19 }
20
21 /* 写/proc 文件接口 */
22 ssize_t simple_proc_write(struct file *filp, const char __user *buff, unsigned
23   long len, void *data)
24 {
25   #define MAX_UL_LEN 8
26   char k_buf[MAX_UL_LEN];
27   char *endp;
28   unsigned long new;
29   int count = min(MAX_UL_LEN, len);
30   int ret;
31
32   if (copy_from_user(k_buf, buff, count)) {
33     ret = -EFAULT;
34     goto err;
35   } else {
36     new = simple_strtoul(k_buf, &endp, 16); /* 字符串转化为整数 */
37     if (endp == k_buf) { /* 无效的输入参数 */
38       ret = -EINVAL;
39       goto err;
40     }
41     val = new;
42     return count;
43   }
44   err:
45   return ret;
46 }
47
48 int __init simple_proc_init(void)
49 {
50   proc_entry = create_proc_entry("sim_proc", 0666, NULL);
51   if (proc_entry == NULL) {
52     printk(KERN_INFO "Couldn't create proc entry\n");

```



```
53     goto err;
54 } else {
55     proc_entry->read_proc = simple_proc_read;
56     proc_entry->write_proc = simple_proc_write;
57     proc_entry->owner = THIS_MODULE;
58 }
59 return 0;
60 err:
61 return -ENOMEM;
62 }
63
64 void __exit simple_proc_exit(void)
65 {
66     remove_proc_entry("sim_proc", &proc_root); //撤销/proc
67 }
68
69 module_init(simple_proc_init);
70 module_exit(simple_proc_exit);
71
72 MODULE_AUTHOR("Barry Song, author@linuxdriver.cn");
73 MODULE_DESCRIPTION("A simple Module for showing proc");
74 MODULE_VERSION("V1.0");
```

上述代码第 36 行调用的 `simple_strtoul()` 用于转换用户输入的字符串为无符号长整数, 第 3 个参数 16 意味着转化方式是十六进制。

编译上述简单的“sim_proc.c”为“sim_proc.ko”, 运行“insmod sim_proc.ko”加载该模块后, /proc 目录下将多出一个文件 sim_proc, “ls -l”的结果如下:

```
[root@localhost proc]# ls -l sim_proc
-rw-rw-rw- 1 root root 0 Sep 4 20:31 sim_proc
```

权限与创建/proc/sim_proc 时给出的 0666 参数是一致的, 现在读取 sim_proc, 如下所示:

```
[root@localhost proc]# cat sim_proc
12345678
```

读出来的正好是我们赋的初值 0x12345678。

测试写/proc/sim_proc 文件, 使用 echo 命令修改它为 0x88888888:

```
[root@localhost driver_study]# echo 88888888 > /proc/sim_proc
```

再查看新值:

```
[root@localhost driver_study]# cat /proc/sim_proc
88888888
```

说明我们上一步执行的写操作是正确的。

22.6 Oops

当内核出现 Segmentation Fault 时 (例如内核访问一个并不存在的虚拟地址), Oops 会被打印到控制台和写入系统 ring buffer。

我们编写一个字符设备驱动, 使让它产生 Oops, 在其读写函数中都访问 0 地址, 如代码清单 22.8 所示。

代码清单 22.8 产生 Oops 设备驱动的读写函数

```

1 static ssize_t oopsexam_read(struct file *filp, char *buf, size_t len, loff_t *off)
2 {
3     int *p=0;
4     *p = 1; /* 故意访问 0 地址 */
5     return len;
6 }
7
8 static ssize_t oopsexam_write(struct file *filp, const char *buf, size_t len, loff_t
9     *off)
10 {
11     int *p=0;
12     *p = 1; /* 故意访问 0 地址 */
13     return len;
14 }

```

假设这个字符设备对应的设备节点是/dev/oops_example，通过“echo 1 > /dev/oops_example”命令写设备文件，将得到如下 Oops 信息：

```

Unable to handle kernel NULL pointer dereference at virtual address 00000000
printing eip:
c381a013
*pde = 00000000
Oops: 0002 [#1]
PREEMPT SMP
Modules linked in: oops_example
CPU: 0
EIP: 0060:[<c381a013>] Not tainted VLI
EFLAGS: 00010286 (2.6.15.5)
EIP is at oopsexam_write+0x4/0x11 [oops_example]
eax: 00000002 ebx: c2b35480 ecx: 00000000 edx: c381a00f
esi: 00000002 edi: 080e9408 ebp: c2007fa4 esp: c2007f68
ds: 007b es: 007b ss: 0068
Process bash (pid: 2453, threadinfo=c2006000 task=c2021570)
Stack: c015e036 c2b35480 080e9408 00000002 c2007fa4 00000000 c2b35480 ffffffff7
080e9408 c2006000 c015e1d1 c2b35480 080e9408 00000002 c2007fa4 00000000
00000000 00000000 00000001 00000002 c0102f9f 00000001 080e9408 00000002
Call Trace:
[<c015e036>] vfs_write+0xc5/0x18f
[<c015e1d1>] sys_write+0x51/0x80
[<c0102f9f>] sysenter_past_esp+0x54/0x75
Code: Bad EIP value.

```

上述 Oops 的第一行给出了“原因”，即访问了“NULL pointer”。Oops 中的“EIP is at oopsexam_write+0x4/0x11 [oops_example]”这一行也比较关键，给出了“事发现场”，即 oopsexam_write() 函数偏移 4 字节的指令处。

通过反汇编可以知道偏移 4 字节的指令对应的 C 代码，如下所示：

```

1 00000000 <oopsexam_read>:
2 0: 8b 44 24 0c          mov     0xc(%esp,1),%eax
3 4: c7 05 00 00 00 01    movl    $0x1,0x0
4 b: 00 00 00
5 e: c3                  ret

```

第 3 行的“movl \$0x1,0x0”对应“*p = 1;”。这里仅仅给出了一个例子，实际的“事发现场”并不这么容易被找到，但方法都是类似的。



同样地, 我们通过“cat /dev/oops_example”命令去读设备文件, 将得到如下 Oops 信息:

```
Unable to handle kernel NULL pointer dereference at virtual address 00000000
printing eip:
c381a004
*pde = 00000000
Oops: 0002 [#2]
PREEMPT SMP
Modules linked in: oops_example
CPU: 0
EIP: 0060:[<c381a004>] Not tainted VLI
EFLAGS: 00010286 (2.6.15.5)
EIP is at oopsexam_read+0x4/0xf [oops_example]
eax: 00001000 ebx: c0d80180 ecx: 00000000 edx: c381a000
esi: 00001000 edi: 0804d8d0 ebp: c1df5fa4 esp: c1df5f68
ds: 007b es: 007b ss: 0068
Process cat (pid: 2969, threadinfo=c1df4000 task=c2021570)
Stack: c015dd9e c0d80180 0804d8d0 00001000 c1df5fa4 00000000 c0d80180 ffffffff
      0804d8d0 c1df4000 c015e151 c0d80180 0804d8d0 00001000 c1df5fa4 00000000
      00000000 00000000 00000003 00001000 c0102f9f 00000003 0804d8d0 00001000
Call Trace:
[<c015dd9e>] vfs_read+0xc5/0x18f
[<c015e151>] sys_read+0x51/0x80
[<c0102f9f>] sysenter_past_esp+0x54/0x75
Code: Bad EIP value.
```

现在给出的“原因”与写时完全相同, 但是“事发地”变成了 oopsexam_read() 函数偏移 4 字节的指令处。

在驱动中如果发现硬件或软件的运行情况与预期的不一致, 完全可以通过下面的语句故意抛出一个 Oops, 以便于提供 bug 的上下文信息:

```
*(int *)0 = 0);
```

内核中有许多地方调用的“BUG();”语句中的 BUG() 宏通常就被定义为该语句, 它非常像一个内核运行时的断言, 意味着本来不该执行到 BUG() 这条语句, 一旦执行即抛出 Oops。BUG() 还有一个变体叫 BUG_ON(), 只有当括号内的条件成立的时候, 才抛出 Oops。

22.7 监视工具

在 Linux 系统中, strace 是一种相当有效的跟踪工具, 它的主要特点是可以被用来监视系统调用。我们不仅可以用 strace 调试一个新开始的程序, 也可以调试一个已经在运行的程序 (这意味着把 strace 绑定到一个已有的 PID 上)。对于第 6 章的 globalmem 字符设备文件, 以 strace 方式运行如代码清单 22.9 所示的用户空间应用程序 globalmem_test, 运行的结果如下:

```
execve("./globalmem_test", ["/globalmem_test"], [/* 24 vars */]) = 0
...
open("/dev/globalmem", O_RDWR) = 3 /* 打开的/dev/globalmem 的 fd 是 3 */
ioctl(3, FIBMAP, 0) = 0
read(3, 0xbff17920, 200) = -1 ENXIO (No such device or address) /* 读取失败 */
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7f04000
```

```

write(1, "-1 bytes read from globalmem\n", 29-1 bytes read from globalmem
) = 29                      /* 向标准输出设备(fd为1)写入printf中的字符串 */
write(3, "This is a test of globalmem", 27) = 27
write(1, "27 bytes written into globalmem\n", 32-27 bytes written into globalmem
) = 32
...

```

输出的每一行对应一次 Linux 系统调用，其格式为“左边=右边”，等号左边是系统调用的函数名及其参数，右边是该调用的返回值。

代码清单 22.9 用户空间应用程序 globalmem_test

```

1 #include ...
2
3 #define MEM_CLEAR 0x1
4 main()
5 {
6     int fd, num, pos;
7     char wr_ch[200] = "This is a test of globalmem";
8     char rd_ch[200];
9     /* 打开/dev/globalmem */
10    fd = open("/dev/globalmem", O_RDWR, S_IRUSR | S_IWUSR);
11    if (fd != -1) { /* 清除 globalmem */
12        if(ioctl(fd, MEM_CLEAR, 0) < 0)
13            printf("ioctl command failed\n");
14        /* 读 globalmem */
15        num = read(fd, rd_ch, 200);
16        printf("%d bytes read from globalmem\n", num);
17
18        /* 写 globalmem */
19        num = write(fd, wr_ch, strlen(wr_ch));
20        printf("%d bytes written into globalmem\n", num);
21        ...
22        close(fd);
23    }
24 }

```

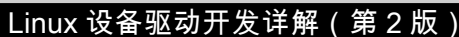
使用 `strace` 虽然无法直接追踪到设备驱动中的函数，但是足够可以帮助工程师推演，如从“`open("/dev/globalmem", O_RDWR) = 3`”的返回结果知道/dev/globalmem 的 fd 为 3，之后对 fd 为 3 的文件进行的 `read()`、`write()`和 `ioctl()`系统调用最终都会引起 globalmem 中 `file_operations` 中的相应函数被调用，通过系统调用的结果就可以知道驱动中 `globalmem_read()`、`globalmem_write()`和 `globalmem_ioctl()`的运行结果。

LDD6410 开发板的文件系统中已经包含了 `strace` 工具，可以直接使用，对应 `strace` 的源代码位于 LDD6410 工程的 `utils/strace-4.5.16` 目录。

22.8 内核调试器

22.8.1 kcore

GDB 调试器可以把内核作为一个应用程序来调试，在这种方式中，需要给 GDB 指定未压缩



```
gdb /usr/src/linux/vmlinux /proc/kcore
```

在“gdb <path>/vmlinux /proc/kcore”这种调试方式中，可用 `print` 命令打印变量，如“`print s`”。当从 GDB 打印数据时，GDB 会缓存已经读取的数据，但是由于内核正在执行，各种项在不同时间有不同的值，GDB 的缓存可能导致连续多次读取同一变量得到相同的值。例如，显示 `jiffies`：

```
(gdb) print jiffies
$3 = 153729
(gdb) print jiffies
$4 = 153729
(gdb) print jiffies
$5 = 153729
```

```
(gdb) core-file /proc/kcore
Core was generated by 'ro root=/dev/sdal hdc=ide-scsi'.
#0  0x00000000 in globalmem_fops ()
(gdb) print jiffies
$6 = 178683
```

为了使 Linux 系统中包含 `/proc/kcore` 文件, 必须在编译时包含“`/proc/kcore support`”(如图 22.10 所示), 而为了给 GDB 提供 symbol 信息, 必须设定 `CONFIG_DEBUG_INFO` 选项来编译内核(如图 22.11 所示)。

[illegible]

图 22.10 编译内核包含/proc/kcore 支持

[illegible]

图 22.11 编译内核包含调试信息

在“gdb <path>/vmlinux /proc/kcore”这种调试方式中，GDB 的绝大多数功能都不能使用，如修改内核变量的值、设置断点、单步执行等，而 22.9.2 和 22.9.3 小节将要介绍的 KDB 和 KGDB 方式则可支持这些功能。

值得一提的是，可加载模块的 `symbol` 并未包含在 `vmlinux` 中，必须使用一些辅助方法才能调试模块。Linux 可加载模块是 ELF 格式的可执行映像，它们被分成几个段，有 3 个典型的与模块调试相关的段。

- **.text**: 这个段包含模块的可执行程序代码。
- **.bss/.data**: 这两个段包含模块的变量，在编译时未初始化的变量在**.bss** 中，而被初始化过的变量在**.data** 段里。

当一个模块被加载后，`/sys/module/`目录下会新增一个对应于该模块的目录，如“`insmod globalmem.ko`”后，将生成`/sys/module/globalmem`，在该目录下又包含一个`sections`目录，运行“`ls -a`”命令可以获得该目录下包含的文件：

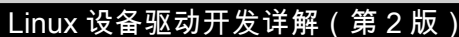
```
[root@localhost sections]# ls -a
.  .bss  .gnu.linkonce.this_module  .rodata.str1.1  .symtab  __versions
.. .data .rodata                .strtab        .text
```

通过 cat 其中的 .text、.data、.bss 可以得到我们感兴趣的 3 个段的地址，如下所示：

```
[root@localhost sections]# cat .text
0xc3816000
[root@localhost sections]# cat .bss
0xc3816b88
[root@localhost sections]# cat .data
0xc3816a94
```

之后就可以借用 GDB 的 `add-symbol-file` 来添加模块的符号信息,这样之后便可以查看模块中的变量了,如下所示:

```
(gdb) add-symbol-file globalmem.ko 0xc3816000\  
-s .bss 0xc3816b88\  
-s .data 0xc3816a94  
add symbol table from file "globalmem.ko" at  
.text_addr = 0xc3816000  
.bss_addr = 0xc3816b88
```

22.8.2 KDB

[illegible]

图 22.12 KDB 编译选项

在引导期间还可以将另一个标志传递给内核，即 `KDB = early`。`early` 标志将导致在引导过程的初始阶段就把控制权传递给 `KDB`，这将非常有利于工程师在引导过程初始阶段进行内核调试。

在两种情况下 KDB 会被调用：当 KDB 处于打开状态时，只要内核中有紧急情况就会自动调用它；其次，按下键盘上的“Pause”键也可手工调用 KDB。使用 KDB 可进行内存和寄存器修改、设置断点和跟踪堆栈等。

KDB 中进行内存显示和修改的常用命令是 `md` 和 `mm`。`md` 命令以一个地址/符号和行计数 `line-count` 为参数，显示从该地址开始的 `line-count` 行的内存。如果没有指定 `line-count`，那么就使用环境变量所指定的默认值。如果没有指定地址，那么 `md` 就从上一次打印的地址继续。`mm` 命令用于修改内存内容，它以地址/符号和新内容作为参数，用新的内容替换指定地址处的内容。

例如，如下命令可显示从 `0xc000000` 开始的 15 行内存：

```
kdb> md 0xc000000 15
```

如下命令可将内存位置为 `0xc000000` 上的内容更改为 `0x10`：

```
kdb> mm 0xc000000 0x10
```

`rd` 命令用于显示处理器寄存器的内容，`rm` 命令用于修改寄存器的内容。它以寄存器名称和新的内容作为参数，用新的内容修改寄存器。寄存器名称与特定的体系结构有关。目前，不能修改控制寄存器。

KDB 中常用的断点命令有 `bp`、`bc`、`bd`、`be` 和 `bl`。`bp` 命令以一个地址/符号作为参数，当遇到该断点时则系统停止执行并将控制权交予 KDB；`bl` 命令列出当前的断点集，它包含了启用和禁用的断点；`be` 命令用于启用断点，该命令的参数是断点号；`bc` 命令用于从断点表中去除断点，它以断点号或“*”作为参数，为“*”意味着去除所有断点。

例如，执行如下命令将对函数 `sys_write()` 设置断点：

```
kdb> bp sys_write
```

执行如下命令可列出断点表中的所有断点：

```
kdb> bl
```

执行如下命令可清除断点号为 1 的断点：

```
kdb> bc 1
```

KDB 中主要的堆栈跟踪命令有 `bt`、`btp`、`btc` 和 `bta`。`bt` 命令提供有关当前线程的堆栈的信息；`btp` 命令以进程标识作为参数，并对这个特定进程进行堆栈回溯；`btc` 命令对每个活动 CPU 上正在运行的进程执行堆栈回溯，它从第一个活动 CPU 开始执行 `bt`，然后切换到下一个活动 CPU，依次类推；`bta` 命令对处于某种特定状态的所有进程执行回溯，可以有选择性地将各种参数传递给该命令，回溯选项包括 `D`（不可中断状态）、`R`（正运行）、`S`（可中断休眠）、`T`（已跟踪或已停止）、`Z`（僵死）和 `U`（不可运行）。`bta` 命令若不带任何参数，会对所有进程执行回溯。

除此之外，在内核调试过程中其他的常用 KDB 命令如下。

- `id` 命令：以地址/符号作为参数，它对从该地址开始的指令进行反汇编，环境变量 `IDCOUNT` 确定要显示输出的行数。
- `ss` 命令：单步执行指令然后将控制返回给 KDB。`ssb` 是该指令的一个变体，它执行从当前指令指针地址开始的指令（在屏幕上打印指令），直到它遇到将引起分支转移的指令为止。
- `go` 命令：让系统继续正常执行，直到遇到断点为止。
- `reboot` 命令：立刻重新引导系统。
- `ll` 命令：以地址、偏移量和另一个 KDB 命令作为参数，它对链表中的每个元素反复执行作为参数的这个命令。



```
kdb> id schedule
```

22.8.3 KGDB

`gdb <path>/vmlinux /proc/kcore` 方式在调试模块时缺少一些至关重要的功能，KDB 尽管克服了部分缺陷，但是它只能在汇编代码级进行调试，而本小节要介绍的 KGDB 则能很方便地在源码级对内核进行调试。KGDB 采用的正是嵌入式系统中远程调试的思路，主机和目标机之间通过串口或网口进行通信。

MontaVista Linux 直接提供了对 KGDB 的支持，而开源社区的内核中必须打上相应版本的 KGDB 补丁（即 kgdb stub，这种方式俗称“插桩”），如 X86 PC 上需要打的补丁包括：core-lite.patch、i386-lite.patch、8250.patch、eth.patch、i386.patch 和 core.patch。KGDB 内核补丁的下载地址为：<http://kgdb.linsyssoft.com/downloads.htm>。

打上 KGDB 补丁后，运行 `make menuconfig` 时需选择关于 KGDB 的编译项目，包括选择“KGDB: kernel debugging with remote gdb”、“KGDB: Console messages through gdb”并设置串口通信方式，如图 22.13 所示。

[illegible]

图 22.13 KGDB 编译选项配置

在 VmWare 中同时启动两个虚拟机就可以模拟 KGDB 的使用环境，在主机上运行如下命令设置串口波特率：

```
stty ispeed 115200 ospeed 115200 -F /dev/ttyS0
```

修改目标机的/etc/grub.conf, 增加如下项目:

```
title Linux-2.6.15.5-kgdb
root (hd0,0)
kernel /boot/vmlinuz-2.6.15.5-kgdb ro root=/dev/sda1 hdc=ide-scsi kgdbwait
```

kgdbwait 的含义是启动时就等待主机的 GDB 连接。

依次运行如下命令就可以启动调试并连接至目标机：

```

<root#> gdb ./vmlinux
GNU gdb Red Hat Linux (6.0post-0.20040223.17rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db library
"/lib/tls/libthread_db.so.1".
(gdb) set remotebaud 115200
(gdb) target remote /dev/ttyS0                //连接目标机
Remote debugging using /dev/ttyS0
breakpoint () at kernel/kgdb.c:1212
1212 atomic_set(&kgdb_setting_breakpoint, 0);
warning: shared library handler failed to enable breakpoint
(gdb)

```

之后，在主机上，我们可以使用 GDB 就像调试应用程序一样调试加载了 KGDB 的目标机上的内核。

为进行可加载模块的调试，需要使用 `gdbmod` 并借助一些技巧来在模块加载的时候获取 `symbol` 信息。首先需要设置 `solib-search-path` 变量的路径，如运行“`set solib-search-path /driver_study`”命令后，再加载 `globalmem.ko`，接着运行“`info sharedlibrary`”命令，如果看到相应的模块信息，就可以在主机上调试加载后的 `globalmem.ko` 模块中的 C 代码了。

最后，需要注意到 KGDB 的工作是以目标系统的串口或网口正常工作为前提的，作为一种软件“插桩”的调试方式，在调试过程中如果出现死机问题，主机上将无法定位。

22.9 使用仿真器调试内核

本节以 BDI2000 为例来讲解如何使用仿真器调试 Linux 内核。BDI2000 是一种最常见的功能强大的仿真器，使用它可以直接调试 Linux 内核。在典型的调试环境中，BDI2000、主机、目标机这 3 者的关系如图 22.14 所示。

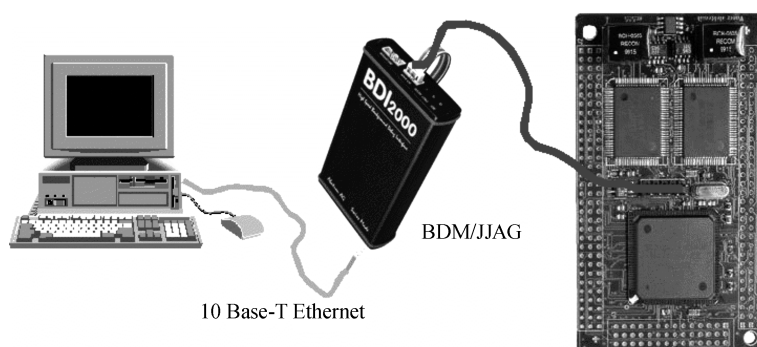


图 22.14 BDI2000、主机与目标机



除了支持免费的 GDB 以外, BDI2000 还支持商业级的 MontaVista DevRocket、LinuxScope (Eclipse GDB) 调试器。

为使用 BDI2000 调试目标板内核, 在 BDI2000 和目标板启动加电后, BDI2000 端需完成如下工作。

(1) 配置 BDI2000, 修改.cfg 文件, 确保目标机的正常初始化。

(2) 主机通过 telnet 登录 BDI2000, 例如, 若主机的/etc/hosts 中包含了“bdi”节点, 则运行“telnet bdi”即可登录到 BDI2000。

(3) 设置内核运行时的第一个断点, 通常在函数 start_kernel(), 如“[BDI2000] bi XXXXXXXX”。通过 grep start_kernel System.map 可以获得 start_kernel() 的具体位置, 代替上面的 XXXXXXXX。

(4) 执行内核代码 “[BDI2000] go”。

以上第 (3)、(4) 步的运行是以目标板已加载 Linux 内核到 RAM 为前提的, 如果没有加载, 则需借助仿真器和主机端加载。

相应地, 主机端需完成如下工作。

(1) 通过 GDB 启动内核调试, 如运行:

```
arm-linux-gdb vmlinux
```

(2) 连接仿真器, 使用 GDB 的“target remote”命令:

```
target remote bdi:2001
```

之后, 就可以在 GDB 中像调试应用程序一样调试目标板上运行的 Linux 内核了。

22.10 应用程序调试

在嵌入式系统中, 为调试 Linux 应用程序, 可在目标板上先运行 GDBServer, 再让主机上的 GDB 与目标板上的 GDBServer 通过网口或串口通信。

1. 目标板

需要运行如下命令启动 GDBServer:

```
gdbserver <host_ip>:<port> <app>
```

<host_ip>:<port>为主机的 IP 地址和端口, app 是可执行的应用程序名。

当然, 也可以用系统中空闲的串口作为 GDB 调试器和 GDBServer 的底层通信手段, 如:

```
gdbserver/dev/ttyS0./tdemo
```

2. 主机

需要先运行如下命令启动 GDB:

```
arm-linux-gdb <app>
```

app 与 GDBServer 的 app 参数对应, arm-linux-gdb 是专门为 ARM 处理器编译出的 GDB 调试器。

之后, 运行如下命令就可以连接目标板:

```
target remote <target_ip>:<port>
```

<target_ip>:<port>为目标机的 IP 地址和端口。

如果目标板上的 GDBServer 使用串口, 则在宿主主机上 GDB 也应该使用串口, 如:

```
(gdb)target remote/dev/ttyS1
```

之后, 便可以使用 GDB 像调试本机上的程序一样调试目标机上的程序。

3. 通过 gdbserver 和 arm-linux-gdb 调试 LDD6410 应用程序

我们为 LDD6410 开发板提供了 gdbserver，下面演示通过以太网口调试 LDD6410 上的应用程序。要调试的应用程序的源代码如下：

```
/*
 * gdb_example.c: program to show how to use arm-linux-gdb
 */

void increase_one(int *data)
{
    *data = *data + 1;
}

int main(int argc, char *argv[])
{
    int dat = 0;
    int *p = 0;
    increase_one(&dat);
    /* program will crash here */
    increase_one(p);
    return 0;
}
```

通过 debug 方式编译它：

```
arm-linux-gcc -g -o gdb_example gdb_example.c
```

将程序下载到目标板后，在目标板上运行：

```
# gdbserver 192.168.1.20:1234 gdb_example
Process gdb_example created; pid = 1096
Listening on port 1234
```

其中 192.168.1.20 为目标板的 IP，1234 为 gdbserver 的侦听端口。

在主机上运行：

```
lihacker@lihacker-laptop:~/ldd6410/tests/gdb-example$ arm-linux-gdb gdb_example
GNU gdb 6.6
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are welcome
to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "--host=/usr/local/arm/4.2.2-eabi/usr/bin/ --target= arm-
linux" ... (gdb)
```

主机的 arm-linux-gdb 中运行如下命令连接目标板：

```
(gdb) target remote 192.168.1.20:1234
Remote debugging using 192.168.1.20:1234
...
0x400007b0 in ?? ()
```

运行如下命令将断点设置在 increase_one(&dat);这一行：

```
(gdb) b gdb_example.c:16
Breakpoint 1 at 0x8390: file gdb_example.c, line 16.
```

通过“c”命令继续运行目标板上的程序，发生断点：

```
(gdb) c
Continuing.
...
Breakpoint 1, main (argc=1, argv=0xbead4eb4) at gdb_example.c:16
16  increase_one(&dat);
```



运行“n”命令执行完 `increase_one(&dat);`

再查看 `dat` 的值:

```
(gdb) n
19  increase_one(p); (gdb) p dat
$1 = 1
```

发现 `dat` 变成 1。继续运行“c”命令, 由于即将访问空指针, `gdb_example` 将崩溃:

```
(gdb) c
Continuing.
Program received signal SIGSEGV, Segmentation fault.
0x0000834c in increase_one (data=0x0) at gdb_example.c:8
8  *data = *data + 1;
```

我们通过“bt”命令可以拿到 `backtrace`:

```
(gdb) bt
#0  0x0000834c in increase_one (data=0x0) at gdb_example.c:8
#1  0x000083a4 in main (argc=1, argv=0xbead4eb4) at gdb_example.c:19
```

通过“`info reg`”命令可以查看当时的寄存器值:

```
(gdb) info reg
r0  0x0 0
r1  0xbead4eb4 3199028916
r2  0x1 1
r3  0x0 0
r4  0x4001e5e0 1073866208
r5  0x0 0
r6  0x826c 33388
r7  0x0 0
r8  0x0 0
r9  0x0 0
r10 0x40025000 1073893376
r11 0xbead4d44 3199028548
r12 0xbead4d48 3199028552
sp  0xbead4d30 0xbead4d30
lr  0x83a4 33700
pc  0x834c 0x834c <increase_one+24>
fps 0x0 0
cpsr 0x60000010 1610612752
```

22.11 Linux 性能监控与调优工具

除了保证程序的正确性以外, 项目开发中往往还关心性能和稳定性。这时候, 我们往往要对内核、应用程序或整个系统进行性能优化。性能优化中常用的手段如下。

1. 使用 `top`、`vmstat`、`iostat`、`sysctl` 等常用工具

`top` 命令显示处理器的活动状况。缺省情况下, 显示占用 CPU 最多的任务, 并且每隔 5s 做一次刷新; `iostat` 命令分析各个磁盘的传输闲忙状况; `vmstat` 命令报告关于内核线程、虚拟内存、磁盘、陷阱和 CPU 活动的统计信息; `netstat` 是用来检测网络信息的工具; `sar` 用于收集、报告或者保存系统活动信息, `sar` 显示数据、`sar1` 和 `sar2` 用于收集和保存数据。

`sysctl` 是一个可用于改变正在运行中的 Linux 系统的接口。用 `sysctl` 可以读取设置超过几百个

系统变量，例如“`sysctl -a`”会读取所有变量。

`sysctl` 的实现原理是：所有的内核参数在 `/proc/sys` 形成一个树状结构，`sysctl` 系统调用调用到的内核函数是 `sys_sysctl`，匹配项目后，最后的读写在 `do_sysctl_strategy` 中完成，如

```
echo "1" > /proc/sys/net/ipv4/ip_forward
```

就等价于：

```
sysctl -w net.ipv4.ip_forward = "1"
```

2. 使用高级分析手段，如 OProfile、gprof

OProfile 可以帮助用户识别诸如模块的占用时间、循环的展开、高速缓存的使用率低、低效的类型转换和冗余操作、错误预测转移等问题。它收集有关处理器事件的信息，其中包括 TLB 的故障、停机、存储器访问以及 `cache` 命中和未命中的指令的攫取数量。

OProfile 支持两种采样方式：基于事件的采样（`event based`）和基于时间的采样（`time based`）。基于事件的采样是 OProfile 只记录特定事件（比如 L2 `cache miss`）的发生次数，当达到用户设定的定值时 `oprofile` 就记录一下（采一个样）。这种方式需要 CPU 内部有性能计数器（`performance counter`）。基于时间的采样是 OProfile 借助 OS 时钟中断的机制，每个时钟中断 OProfile 都会记录一次（采一次样）。引入的目的在于，提供对没有性能计数器 CPU 的支持。其精度相对于基于事件的采样要低。因为要借助 OS 时钟中断的支持，对禁用中断的代码 OProfile 不能对其进行分析。

OProfile 在 Linux 上分两部分，一个是内核模块（`oprofile.ko`），一个为用户空间的守护进程（`oprofiled`）。前者负责访问性能计数器或者注册基于时间采样的函数，并采样置于内核的缓冲区内。后者在后台运行，负责从内核空间收集数据，写入文件。其运行步骤如下。

- (1) 初始化 `opcontrol --init`
- (2) 配置 `opcontrol --setup --event=...`
- (3) 启动 `opcontrol --start`
- (4) 运行待分析之程序 `xxx`
- (5) 取出数据

`opcontrol --dump`
`opcontrol --stop`
- (6) 分析结果 `opreport -l ./xxx`

GNU `gprof` 可以打印出程序运行中各个函数消耗的时间，可以帮助程序员找出众多函数中耗时最多的函数；产生程序运行时候的函数调用关系，包括调用次数，可以帮助程序员分析程序的运行流程。

GNU `gprof` 的实现原理为通过在编译和链接程序的时候（使用 `-pg` 编译和链接选项），`gcc` 在应用程序的每个函数中都加入名为 `mcount`（或“`_mcount`”，或“`__mcount`”，依赖于编译器或操作系统）的函数，也就是说应用程序里的每一个函数都会调用 `mcount`，而 `mcount` 会在内存中保存一张函数调用图，并通过函数调用堆栈的形式查找子函数和父函数的地址。这张调用图也保存了所有与函数相关的调用时间，调用次数等的信息。

GNU `gprof` 的基本用法如下。

- (1) 使用 `-pg` 编译和链接应用程序。



(2) 执行应用程序使之生成供 gprof 分析的数据。

(3) 使用 gprof 程序分析应用程序生成的数据。

3. 进行内核跟踪, 如 LTT

LTT (Linux Trace Toolkit) 是一个用于跟踪系统详细运行状态和流程的工具, 它可以跟踪记录系统中的特定事件。这些事件包括: 系统调用的进入和退出; 陷阱/中断 (trap / irq) 的进入和退出; 进程调度事件; 内核定时器; 进程管理相关事件: 创建、唤醒、信号处理等; 文件系统相关事件: open/read/write /seek / ioctl 等; 内存管理相关事件: 内存分配/释放等; 其他 IPC/socket/网络等事件。而这些记录我们可以通过图形的方式查看, 如图 20.15 所示。

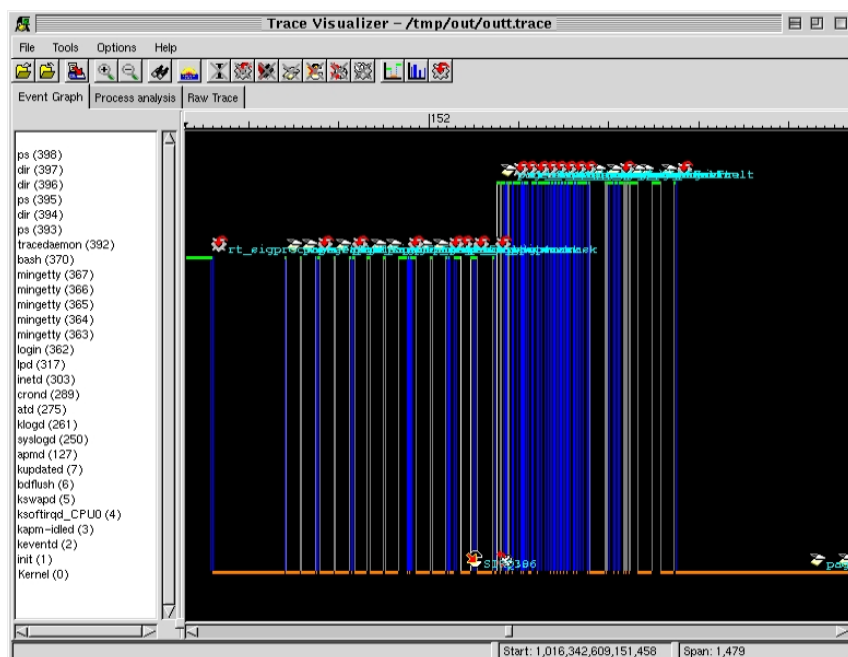


图 20.15 LTT 形成的时序图

4. 使用 LTP 进行压力测试

LTP (Linux Test Project, 官方网站 <http://ltp.sourceforge.net/>) 是一个由 SGI 发起并由 IBM 负责维护的合作计划。它的目的是为开源社区提供测试套件来验证 Linux 的可靠性、健壮性和稳定性。它通过压力测试来判断系统的稳定性和可靠性, 工程中我们可使用 LTP 测试套件对 Linux 操作系统进行超长时间的测试, 它可进行文件系统压力测试、硬盘 I/O 测试、内存管理压力测试、IPC 压力测试、SCHED 测试、命令功能的验证测试、系统调用功能的验证测试等。

5. 使用 benchmark 评估系统系统

可用于 Linux 的 benchmark 包括 Imbench、UnixBench、AIM9、Netperf、SSLperf、dbench、Bonnie、Bonnie++、Iozone、BYTEmark 等, 用于评估操作系统、网络、IO 子系统、CPU 等的性能, 参考网址 <http://lts.sourceforge.net/> 列出了许多 benchmark 工具。

22.12 总结

Linux 程序的调试尤其是内核的调试看起来比较复杂，没有类似于 VC++、Tornado 的 IDE 开发环境，最常用的调试手段依然是文本方式的 GDB。文本方式的 GDB 调试器功能异常强大，当我们使用习惯后，就会用得非常自然了。

Linux 内核驱动的调试方法包括“插桩”、使用仿真器和借助 `printk()`、`oops`、`strace` 等，在大多数情况下，原始的 `printk()` 仍然是最有效的手段。

除了本章介绍的方法外，在驱动的调试中很可能还会借助其他的硬件或软件调试工具，如调试 USB 驱动最好借助 USB 分析仪，USB 分析仪将可捕获 USB 通信中的包，如同网络中的 sniffer 软件一样。

LINUX

第23章 Linux 设备驱动的移植

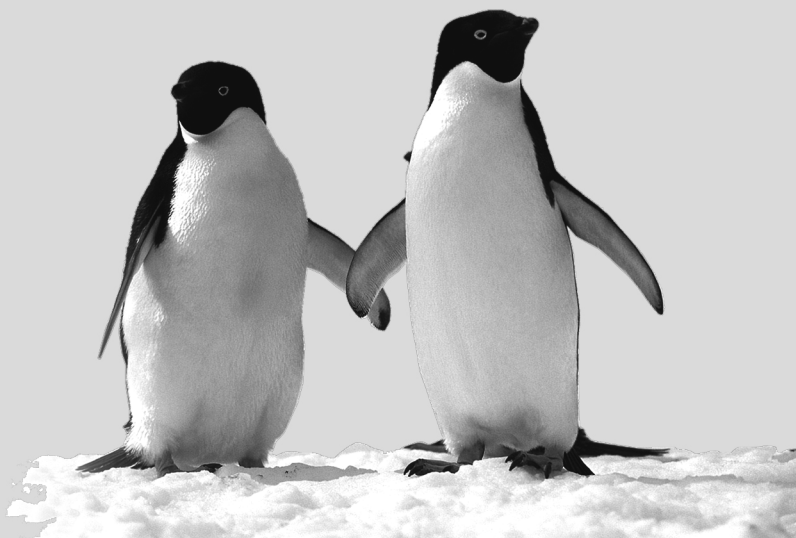
在编写 Linux 设备驱动的时候，驱动程序所服务的硬件芯片可能会在公司的多个采用不同处理器的系统中用到，因此，在编写驱动时，应该尽量考虑其可移植性，23.1 节从数据类型、结构体对界、大小端模式、内存页面大小等多个角度阐述了编写可移植驱动程序的注意事项。

“他山之石，可以攻玉”，为了高效地推出新的设备驱动，借用 demo 板、类似芯片和厂商范例程序是几种很有效的手段，23.2 节讲解了这些快速编写设备驱动的方法。

23.3 节讲解了 Linux 2.4 和 Linux 2.6 内核在设备驱动方面的差异，通过对两者差异的分析，可以得出移植 Linux 2.4 内核驱动到 Linux 2.6 内核的方法。

23.4 节给出了将其他操作系统内的驱动移植到 Linux 中的方法，主要分析了实时操作系统 VxWorks 设备驱动和 Linux 设备驱动的异同点。

23.5 讲解了如何将 Linux 移植到新的 SoC 和电路板。



23.1 编写可移植的设备驱动

23.1.1 可移植的数据类型

C 语言中的标准数据类型 `int`、`long` 的长度直接与平台相关，在驱动中，关键部分代码直接使用这些类型时需要特别小心。表 23.1 给出了在几个不同的平台下 `char`、`short`、`int`、`long`、`ptr`、`long long` 的长度。

表 23.1 不同平台下 `char`、`short`、`int`、`long`、`ptr`、`long long` 的长度

arch	char	short	int	long	ptr	long long
i386	1	2	4	4	4	8
Alpha	1	2	4	8	8	8
armv4l	1	2	4	4	4	8
ia64	1	2	4	8	8	8
M68K	1	2	4	4	4	8
MIPS	1	2	4	4	4	8
PowerPC	1	2	4	4	4	8
Sparc	1	2	4	4	4	8
Sparc64	1	2	4	4	4	8
x86_64	1	2	4	8	8	8

因此，在 Linux 系统中，针对不同的体系结构重新 `typedef` 出了 `u8`、`u16`、`u32`、`u64`、`s8`、`s16`、`s32`、`s64` 等类型。例如，在 i386/arm 下这些类型的定义如代码清单 23.1 所示，而在 ppc64 下这些类型的定义则如代码清单 23.2 所示，可见影响 C 语言基本数据类型大小的主要因素是 CPU 字长。

代码清单 23.1 i386/arm 平台下 `u8`、`u16`、`u32`、`u64`、`s8`、`s16`、`s32`、`s64` 的定义

```

1 typedef signed char s8;
2 typedef unsigned char u8;
3
4 typedef signed short s16;
5 typedef unsigned short u16;
6
7 typedef signed int s32;
8 typedef unsigned int u32;
9
10 typedef signed long long s64;
11 typedef unsigned long long u64;
```

代码清单 23.2 ppc64 平台下 `u8`、`u16`、`u32`、`u64`、`s8`、`s16`、`s32`、`s64` 的定义

```

1 typedef signed char s8;
2 typedef unsigned char u8;
3
4 typedef signed short s16;
```



```
5 typedef unsigned short u16;
6
7 typedef signed int s32;
8 typedef unsigned int u32;
9
10 typedef signed long s64;
11 typedef unsigned long u64;
```

由于 u8、u16、u32、u64、s8、s16、s32、s64 是被针对不同的体系结构单独定义的, 因此, 在任何平台下对其进行 sizeof 运算的结果都是不变的, 是确定长度的数据类型。但是, 这些类型都应该只在内核空间使用。在 Linux 用户空间中, 如果要使用确定长度的数据类型, 应该使用上述定义加 “__” 的版本, 如 __u8、__u16、__u32 等。鉴于此, 设备驱动中如果要向用户空间导出头文件, 在头文件中也应该定义 __sxx、__uxx 等数据类型。

上面给出的 uxx、sxx、__uxx、__sxx 类型定义都是 Linux 系统所特有的, 为了更好地向其他平台移植, 驱动中最好使用 int8_t、int16_t、int32_t、uint8_t、uint16_t、uint32_t、int64_t、uint64_t 这些 C99 标准确定长度类型。

Linux 系统中定义了许多以 _t 为后缀的数据类型, 这些类型用在内核的一些常用功能的实现中, 如 dma_addr_t、uid_t、gid_t、size_t、ssize_t、pid_t、loff_t 等, 这些类型的使用将使内核屏蔽实际数据类型间存在的差异。例如, file_operations 中的 read()、write() 成员函数返回值为 ssize_t 类型, llseek() 返回的是 loff_t 类型。此外, ssize_t、pid_t 这些类型也被赋予了一些含义, 这一点从名字就可以看出来, 如 pid_t 是进程 ID 类型。

23.1.2 结构体对界

在 C 语言中使用结构体时有一个需要特别注意的事项, 那就是结构体的对界。struct 是一种复合数据类型, 其构成元素既可以是基本数据类型的变量, 也可以是一些复合数据类型 (如数据、结构体、联合体等) 的数据单元。对于结构体, 编译器很可能会自动进行成员变量的对齐, 以提高存取效率。默认情况下, 编译器为结构体的每个成员按其自然对界 (natural alignment) 条件分配空间。各个成员按照它们被声明的顺序在内存中顺序存储, 第一个成员的地址和整个结构的地址相同。

自然对界指按结构体的成员中 sizeof 最大的成员对齐 (如果 sizeof 大于 CPU 的字长, 仍然按照 CPU 字长对齐), 例如对于 32 位系统:

```
struct naturalalign {
    char a;
    short b;
    char c;
};
```

在上述结构体中, size 最大的是 short, 其长度为两个字节, 因而结构体中的 char 成员 a、c 都以 2 为单位对齐, sizeof(naturalalign) 的结果等于 6。

如果改为:

```
struct naturalalign {
    char a;
    int b;
    char c;
};
```

其结果为 12。

在 Linux 内核编程中，为了防止编译器自动在结构体的数据间插入空隙，可以使用 `__attribute__((packed))` 定义结构体，如：

```
struct {
    u16 id;
    u64 lun;
    u16 reserved1;
    u32 reserved2;
} __attribute__((packed)) scsi; //不要在数据间插入空隙
```

23.1.3 Little Endian 与 Big Endian

采用 Little Endian 模式的 CPU 对操作数的存放方式是从低字节到高字节，而 Big Endian 模式对操作数的存放方式是从高字节到低字节。例如，16bit 宽的数 0x1234 在 Little Endian 模式 CPU 内存中的存放方式（假设从地址 0x4000 开始存放）为：

内存地址	0x4000	0x4001
存放内容	0x34	0x12

而在 Big Endian 模式，CPU 内存中的存放方式则为：

内存地址	0x4000	0x4001
存放内容	0x12	0x34

32bit 宽的数 0x12345678 在 Little Endian 模式 CPU 内存中的存放方式（假设从地址 0x4000 开始存放）为：

内存地址	0x4000	0x4001	0x4002	0x4003
存放内容	0x78	0x56	0x34	0x12

而在 Big Endian 模式 CPU 内存中的存放方式则为：

内存地址	0x4000	0x4001	0x4002	0x4003
存放内容	0x12	0x34	0x56	0x78

内核中定义如下多个宏来进行 Big Endian 模式与 Little Endian 模式的互换，包括 `cpu_to_le64`、`le64_to_cpu`、`cpu_to_le32`、`le32_to_cpu`、`cpu_to_le16`、`le16_to_cpu`。

内核中定义如下多个宏来进行 Big Endian 模式与 Big Endian 模式的互换：

```
cpu_to_be64、be64_to_cpu、cpu_to_be32、be32_to_cpu、cpu_to_be16、be16_to_cpu。
```

在 Linux/drivers 目录的 ATM、IEEE1394、SCSI、NET、USB 等源码中，都大量存在对上述这些宏的使用。

23.1.4 内存页面大小

一般情况下，内存页面的大小是 4KB（即 `PAGE_SIZE` 定义为 4KB），但是这并非是一定的，实际上，页面大小在一个 4~64KB 的范围内是可变的，即使在相同的平台下也可以定义不同的 `PAGE_SIZE` 和 `PAGE_SHIFT`。



鉴于此,当在内核空间中通过 `get_free_pages()` 函数申请内存时,如果它的第二个参数为 `order`,意味着申请 $\text{PAGE_SIZE} * 2^{\text{order}}$ 的内存,同样是申请 64KB 内存,如果 `PAGE_SIZE` 为 4KB,应该传入的 `order` 是 4,如果 `PAGE_SIZE` 是 16KB,应该传入的参数是 2。为了保证在申请 64KB 内存时,在任何 `PAGE_SIZE` 的情况下都成立,可以使用如代码清单 23.3 所示的方法。

代码清单 23.3 通过 `get_order()` 获得要申请内存的 `order`

```
1 #include <asm/page.h>
2 int order = get_order(64*1024);
3 buf = get_free_pages(GFP_KERNEL, order);
```

23.2 巧用同类设备驱动

23.2.1 巧用 demo 板驱动

对于推出的重要芯片,芯片厂商往往会同时提供一套 demo 板。这样的 demo 板不仅在硬件设计中被硬件工程师充分利用并进行参考,而且其提供的驱动程序往往在新设计的硬件系统中被参考。

借用 demo 板驱动的方法主要是寻找共性中的差异,例如共性是芯片相同,差异则可能体现在所使用的 I/O 内存(片选)、中断和 DMA 资源不同,在这种情况下,简单地修改 I/O 内存基地址、中断号以及 DMA 通道, demo 板的驱动就可以用在目标电路板上。而如果除了芯片相同以外,外围芯片与 CPU 连接所用的内存、中断和 DMA 资源都相同的话,则 demo 板驱动基本上可以不加任何修改地搬到目标电路板上。

如果 demo 板和目标电路板所用资源不同,而 demo 板对应设备被定义为 `platform_device`,且其资源并定义在 `resource` 结构体数组中,则直接修改 `resource` 结构体即可,如下所示:

```
static struct resource xxx_resource[] = {
    [0] = {
        .start = XXX_MEM_START, /* 修改这里替换 I/O 内存基地址 */
        .end   = XXX_MEM_START + XXX_MEM_SIZE,
        .flags = IORESOURCE_MEM,
    },

    [1] = {
        .start = XXX_INT_START, /* 修改这里换中断 */
        .end   = XXX_INT_END,
        .flags = IORESOURCE_IRQ,
    }
};
```

同时,我们在编写新驱动的时候永远要牢记这样的设计理念:将硬件和平台相关的信息(内存地址、中断号、DMA 通道、硬件设置等)放入 BSP 中,作为 platform 信息、SPI board 信息、I²C board 信息等,而不是直接放在驱动里面。

23.2.2 巧用类似芯片的驱动程序

任何驱动工程师都没有必要在面对新设备驱动编写需求的时候一切从头开始,因为内核源代

码 drivers 目录（音频设备的驱动在 sound 目录）中已经包含了大量现成的类似芯片驱动的源代码，是极好的参考模板，我们不需要“re-invent the wheel”。实际上，在内核源代码许多后期编写的驱动程序中，就直接参考了之前的驱动源码，所以同类设备的驱动往往呈现出非常相似的架构和数据结构定义。我们来看看 sound/oss/ au1550_ac97.c 文件最开始的一段注释：

```
/*
 * au1550_ac97.c -- Sound driver for Alchemy Au1550 MIPS Internet Edge
 *                 Processor.
 *
 * Copyright 2004 Embedded Edge, LLC
 *   dan@embeddededge.com
 *
 * Mostly copied from the au1000.c driver and some from the
 * PowerMac dbdma driver.
 * We assume the processor can do memory coherent DMA.
 * ...
 */
```

这段注释很清楚地说明其绝大多数代码都来自 au1000.c 驱动，还有一些来自 PowerMac dbdma 驱动。

打开 sound/oss 目录下的 au1550_ac97.c（Alchemy Au1550 MIPS 处理器的音频驱动）、es1370.c（Ensoniq ES1370/Asahi Kasei AK4531 声卡驱动）、es1371.c（reative Ensoniq ES1371 声卡驱动）、cs46xx.c（Crystal SoundFusion CS46xx 声卡驱动），发现如下相似之处。

- 它们全都自定义了全局的 xxx_state 结构体实例用于封装音频设备的锁、信号量、缓冲区分、ID 等信息，这几个结构体分别是：au1550_state、es1370_state、es1371_state、cs_state。
- 它们的核心函数都使用了完全相同的实现方法，au1550_ac97.c 的 au1550_read() 和 es1370.c 的 es1370_read() 的处理流程是一致的，下面列表的左右两列对等地给出了 au1550_read() 和 es1370_read() 函数的源代码（为了进行横向比较，适当地增加了源代码的换行，以达到类似 WinMerge 等源码比较软件的效果）。

```
static ssize_t u1550_read(struct file
 *file, char *buffer, size_t count,
 loff_t *ppos)
{
    struct au1550_state *s = (struct
        au1550_state *) file->private_data;
    struct dmabuf *db = &s->dma_adc;
    DECLARE_WAITQUEUE(wait, current);
    ssize_t ret;
    unsigned long flags;
    int cnt, usercnt, avail;

    if (db->mapped)
        return - ENXIO;
    if (!access_ok(VERIFY_WRITE, buffer,
        count))
        return - EFAULT;
    ret = 0;
```

```
Static ssize_t es1370_read(struct file
 *file, char __user *buffer, size_t
 count, loff_t *ppos)
{
    struct es1370_state *s = (struct
        es1370_state *) file->private_data;
    DECLARE_WAITQUEUE(wait, current);
    ssize_t ret = 0;
    unsigned long flags;
    unsigned swptr;
    int cnt;

    VALIDATE_STATE(s);
    if (s->dma_adc.mapped)
        return - ENXIO;
    if (!access_ok(VERIFY_WRITE, buffer,
        count))
        return - EFAULT;
```




```
count *= db->cnt_factor;

down(&s->sem);

add_wait_queue(&db->wait, &wait);

while (count > 0)
{
    /* wait for samples in ADC dma buffer
    */
    do
    {
        spin_lock_irqsave(&s->lock, flags);
        if (db->stopped)
            start_adc(s);
        avail = db->count;

        if (avail <= 0)
            _set_current_state
                (TASK_INTERRUPTIBLE);
        spin_unlock_irqrestore(&s->lock,
            flags);

        if (avail <= 0)
        {
            if (file->f_flags & O_NONBLOCK)
            {
                if (!ret)
                    ret = -EAGAIN;
                goto out;
            }
            up(&s->sem);
            schedule();
            if (signal_pending(current))
            {
                if (!ret)
                    ret = -ERESTARTSYS;
                goto out2;
            }
            down(&s->sem);
        }
    }
    while (avail <= 0);

    /* copy from nextOut to user
    */

    if ((cnt = copy_dmabuf_user(db,
```

```
down(&s->sem);
if (!s->dma_adc.ready && (ret =
    prog_dmabuf_adc(s)))
    goto out;

add_wait_queue(&s->dma_adc.wait, &wait);

while (count > 0)
{

    spin_lock_irqsave(&s->lock, flags);
    swptr = s->dma_adc.swptr;
    cnt = s->dma_adc.dmasize - swptr;
    if (s->dma_adc.count < cnt)
        cnt = s->dma_adc.count;
    if (cnt <= 0)
        _set_current_state
            (TASK_INTERRUPTIBLE);
    spin_unlock_irqrestore(&s->lock,
        flags);
    if (cnt > count)
        cnt = count;
    if (cnt <= 0)
    {
        if (s->dma_adc.enabled)
            start_adc(s);
        if (file->f_flags & O_NONBLOCK)
        {
            if (!ret)
                ret = -EAGAIN;
            goto out;
        }
        up(&s->sem);
        schedule();
        if (signal_pending(current))
        {
            if (!ret)
                ret = -ERESTARTSYS;
            goto out;
        }
        down(&s->sem);
        if (s->dma_adc.mapped)
        {
            ret = -ENXIO;
            goto out;
        }
        continue;
    }
    if (copy_to_user(buffer, s
```

<pre> buffer, count > avail ? avail : count, 1)) < 0) { if (!ret) ret = - EFAULT; goto out; } spin_lock_irqsave(&s->lock, flags); db->count -= cnt; db->nextOut += cnt; if (db->nextOut >= db->rawbuf + db ->dmasize) db->nextOut -= db->dmasize; spin_unlock_irqrestore(&s->lock, flags); count -= cnt; usercnt = cnt / db->cnt_factor; buffer += usercnt; ret += usercnt; } /* while (count > 0) */ out: up(&s->sem); out2: remove_wait_queue(&db->wait, &wait); set_current_state(TASK_RUNNING); return ret; } </pre>	<pre> ->dma_adc.rawbuf + swptr, cnt)) { if (!ret) ret = - EFAULT; goto out; } swptr = (swptr + cnt) % s ->dma_adc.dmasize; spin_lock_irqsave(&s->lock, flags); s->dma_adc.swptr = swptr; s->dma_adc.count -= cnt; spin_unlock_irqrestore(&s->lock, flags); count -= cnt; buffer += cnt; ret += cnt; if (s->dma_adc.enabled) start_adc(s); } out: up(&s->sem); remove_wait_queue(&s->dma_adc.wait, &wait); set_current_state(TASK_RUNNING); return ret; } </pre>
---	--

可以看出，内核中看似神秘的、庞大的设备驱动源码也是互相学习、互相借鉴的结果。

23.2.3 借用芯片厂商的范例程序

在外围芯片上市之前，芯片厂商往往进行了严格的验证，在他们的验证过程中，必然会编写代码去访问和控制这些芯片。很多时候，这些代码稍经整理就被芯片厂商随同 **datasheet** 一起在网站上作为参考代码发布。

范例程序往往停留在无操作系统的层次上，只是最底层的硬件操作代码，这一部分代码对驱动工程师的意义如下。

- 帮助工程师进一步理解芯片与 CPU 的接口原理、芯片的访问和控制方法。
- 直接加以改进后搬到 Linux 设备驱动中。

Linux 设备驱动的硬件操作方法会与无操作系统时的硬件操作方法有如下差异。

- 无操作系统的硬件访问方法中往往没有物理地址到虚拟地址的映射过程，因此，在搬到 Linux 系统中的时候，要注意以静态映射或 **ioremap()** 等方式映射到虚拟地址。
- 硬件访问中往往夹杂着延时，因此，在无操作系统的源码中，经常会出现 **xxx_delay()** 这样的 for 循环延迟，这些代码应该被内核中的 **ndelay()** 或 **udelay()** 替换。如果延迟时间达到数十 ms，应该使用 **msleep()** 或 **msleep_interruptible()** 等函数。



- 芯片范例程序只是对芯片的操作方法进行示范, 它并不会考虑真实应用场景中对 CPU 的资源占用以及代码的时间性能。例如, 如果在写寄存器 REGA 后, 要判断寄存器 REGB 的第 0 位为 1 后才能进行下一次写, 则无操作系统中的代码呈现为:

```
write_rega(int value)
{
    rega = value;
    while (!(regb & 0x1));
}
```

第 2 句的 `while (!(regb & 0x1))` 是比较致命的, 如果系统中用的 Linux 不支持抢占调度, 而 REGB 的第 0 位变成 1 需要相当长的时间 (如数十 ms), 这种忙等待会导致其他的进程全部得不到机会执行。即使 Linux 支持抢占调度, 进行这样的忙等待也毫无意义, Linux 中理想的做法是进行在这种情况下调度其他进程执行或者调用 `cpu_relax()`:

```
while (my_variable != what_i_want)
    cpu_relax();
```

其中的 `cpu_relax()` 的作用是降低 CPU 的消耗, 同时也起到内存屏障 (memory barrier) 的作用, 因此在内核的 `Documentation/volatile-considered-harmful.txt` 文档中, 建议这种忙等待也不要使用 `volatile` 关键字。

使用 `while (!(regb & 0x1))` 这样的判断还有一个问题, 如果硬件出现了故障, REGB 的第 0 位总是变不成 1 的话, 在系统不支持抢占调度的情况下, 就“死机”了, 所以在进行忙等待的时候, 许多场合下会设置一个超时机制。

23.3 从 Linux 2.4 移植设备驱动到 Linux 2.6

从 Linux 2.4 内核到 Linux 2.6 内核, Linux 在可装载模块机制、设备模型、一些核心 API 等方面发生了较大改变, 随着公司产品的过渡, 驱动工程师会面临着将驱动从 Linux 2.4 内核移植到 Linux 2.6 内核, 或是让驱动能同时支持 Linux 2.4 内核与 Linux 2.6 内核的任务。

下面分析 Linux 2.4 内核和 Linux 2.6 内核在设备驱动方面的几个主要的不同点。

1. 内核模块的 Makefile

Linux 2.4 内核中, 模块的编译只需内核源码头文件, 并在包括 `linux/modules.h` 头文件之前定义 `MODULE`, 且其编译、连接后生成的内核模块后缀为 `.o`。而在 Linux 2.6 内核中, 模块的编译需要依赖配置过的内核源码, 编译过程首先会到内核源码目录下, 读取顶层的 `Makefile` 文件, 然后再返回模块源码所在目录, 且编译、连接后生成的内核模块后缀为 `.ko`。

Linux 2.4 中内核模块的 `Makefile` 模板如代码清单 23.4 所示。

代码清单 23.4 Linux 2.4 中内核模块的 `Makefile` 模板

```
1 #Makefile2.4
2 KVER=$(shell uname -r)
3 KDIR=/lib/modules/$(KVER)/build
4 OBJS=mymodule.o
5 CFLAGS=-D__KERNEL__ -I$(KDIR)/include -DMODULE -D__KERNEL_SYSCALLS__
6 -DEXPORT_SYMTAB -O2 -fomit-frame-pointer -Wall -DMODVERSIONS
```

```

7  -include $(KDIR)/include/linux/modversions.h
8  all: $(OBSJ)
9    mymodule.o: file1.o file2.o
10   ld -r -o $$ $^
11 clean:
12   rm -f *.o

```

而 Linux 2.6 中内核模块的 Makefile 模板如代码清单 23.5 所示。

代码清单 23.5 Linux 2.6 中内核模块的 Makefile 模板

```

1  # Mcakefile2.6
2  ifneq ($(KERNELRELEASE),)
3  #dependency relationship of files and target modules
4  #mymodule-objs := file1.o file2.o
5  #obj-m := mymodule.o
6  obj-m := second.o
7  else
8  PWD := $(shell pwd)
9  KVER ?= $(shell uname -r)
10 KDIR := /lib/modules/$(KVER)/build
11 all:
12   $(MAKE) -C $(KDIR) M=$(PWD)
13 clean:
14   rm -rf *.cmd *.o *.mod.c *.ko .tmp_versions
15 endif

```

Linux 2.6 内核模板 Makefile 中的 KERNELRELEASE 是在内核源码的顶层 Makefile 中定义的一个变量，在第一次读取执行此 Makefile 时，KERNELRELEASE 没有被定义，所以 make 将读取执行 else 之后的内容。如果 make 的目标是 clean，将直接执行 clean 操作，然后结束；当 make 的目标为 all 时，-C \$(KDIR) 指明跳转到内核源码目录下读取那里的 Makefile，M=\$(PWD) 表明之后要返回到当前目录继续读入、执行当前的 Makefile。当从内核源码目录返回时，KERNELRELEASE 已被定义，kbuild 也被启动去解析 kbuild 语法的语句，make 将继续读取 else 之前的内容。else 之前的内容为 kbuild 语法的语句，指明模块源码中各文件的依赖关系，以及要生成的目标模块名。“mymodule-objs := file1.o file2.o”表示 mymodule.o 由 file1.o 与 file2.o 连接生成，“obj-m := mymodule.o”表示编译连接后将生成 mymodule 模块。

“\$(MAKE) -C \$(KDIR) M=\$(PWD)”与“\$(MAKE) -C \$(KDIR) SUBDIRS=\$(PWD)”的作用是等效的，后者是较老的使用方法。

通过以上比较可以看到，从 Makefile 编写角度来看，在 Linux 2.6 内核下，内核模块编译不必定义复杂的 CFLAGS，而且模块中各文件依赖关系的表示更加简洁清晰。

在分析清楚 Linux 2.4 和 Linux 2.6 的内核模块 Makefile 的差异之后，可以给出同时支持 Linux 2.4 内核和 Linux 2.6 内核的内核模块 Makefile 文件，如代码清单 23.6 所示。这个模板中实际上根据内核版本，去读取不同的 Makefile。

代码清单 23.6 同时支持 Linux 2.4/2.6 的内核模块的 Makefile 模板

```

1  #Makefile for 2.4 & 2.6
2  VERS26=$(findstring 2.6,$(shell uname -r))
3  MAKEDIR?=$(shell pwd)
4  ifeq ($(VERS26),2.6)
5  include $(MAKEDIR)/Makefile2.6

```



```
6 else
7   include $(MAKEDIR)/Makefile2.4
8 endif
```

2. 内核模块加载时的版本检查

Linux 2.4 内核下, 执行 “cat /proc/ksyms”, 将会看到内核符号, 而且在名字后还会跟随着一串校验字符串, 此校验字符串与内核版本有关。在内核源码头文件 `linux/modules` 目录下存在许多 *.ver 文件, 这些文件起着为内核符号添加校验后缀的作用, 如 `ksyms.ver` 文件里有一行 “#define printk_set_ver(printk)”, `linux/modversions.h` 文件会包含所有的 .ver 文件。

所以当模块包含 `linux/modversions.h` 文件后, 编译时, 模块里使用的内核符号实质上成为带有校验后缀的内核符号。在加载模块时, 如果模块使用的内核符号的校验字符串与当前运行内核所导出的相应的内核符号的校验字符串不一致, 即当前内核空间并不存在模块所使用的内核符号, 就会出现 “Invalid module format” 的错误。

Linux 内核所采用的在内核符号添加校验字符串来验证模块的版本与内核的版本是否匹配的方法很复杂且会浪费内核空间, 而且随着 SMP、PREEMPT 等机制在 Linux 2.6 内核的引入和完善, 模块运行时对内核的依赖不再仅仅取决于内核版本, 还取决于内核的配置, 此时内核符号的校验码是否一致不能成为判断模块可否被加载的充分条件。

在 Linux 2.6 内核的 `linux/vermagic.h` 头文件中定义了 “版本魔术字符串” —— VERMAGIC_STRING (如代码清单 23.7 所示), VERMAGIC_STRING 不仅包含内核版本号, 还包含内核编译所使用的 GCC 版本、SMP 与 PREEMPT 等配置信息。在编译模块时, 我们可以看到屏幕上会显示 “MODPOST” (模块后续处理), 在内核源码目录下 `scripts/mod/modpost.c` 文件中可以看到模块后续处理部分的代码。

就是在这个阶段, VERMAGIC_STRING 会被添加到模块的 `modinfo` 段中, 模块编译生成后, 通过 “`modinfo mymodule.ko`” 命令可以查看此模块的 `vermagic` 等信息。

Linux 2.6 内核下的模块装载机里保存有内核的版本信息, 在装载模块时, 装载机会比较所保存的内核 `vermagic` 与此模块的 `modinfo` 段里保存的 `vermagic` 信息是否一致, 两者一致时, 模块才能被装载。

代码清单 23.7 VERMAGIC_STRING 的定义

```
1 #ifdef CONFIG_SMP    /* 配置了 SMP */
2 #define MODULE_VERMAGIC_SMP "SMP "
3 #else
4 #define MODULE_VERMAGIC_SMP ""
5 #endif
6
7 #ifdef CONFIG_PREEMPT    /* 配置了 PREEMPT */
8 #define MODULE_VERMAGIC_PREEMPT "preempt "
9 #else
10 #define MODULE_VERMAGIC_PREEMPT ""
11 #endif
12
13 #ifdef CONFIG_MODULE_UNLOAD    /* 支持 module 卸载 */
14 #define MODULE_VERMAGIC_MODULE_UNLOAD "mod_unload "
15 #else
16 #define MODULE_VERMAGIC_MODULE_UNLOAD ""
```

```

17 #endif
18
19 #ifndef MODULE_ARCH_VERMAGIC /* 体系结构 VERMAGIC */
20 #define MODULE_ARCH_VERMAGIC ""
21 #endif
22
23 /* 拼接内核版本、上述 VERMAGIC 以及 GCC 版本 */
24 #define VERMAGIC_STRING \
25 UTS_RELEASE " " \
26 MODULE_VERMAGIC_SMP MODULE_VERMAGIC_PREEMPT \
27 MODULE_VERMAGIC_MODULE_UNLOAD MODULE_ARCH_VERMAGIC \
28 "gcc-" __stringify(__GNUC__) "." __stringify(__GNUC_MINOR__)

```

在通过 `make menuconfig` 对内核进行新的配置后，再基于 Linux 2.6.15.5 内核编译生成的 `hello.ko` 模块（见第 4 章），这个模块的 `modinfo` 结果如下：

```

[root@localhost driver_study]# modinfo hello.ko
filename:      hello.ko
license:       Dual BSD/GPL
author:        Song Baohua
description:    A simple Hello World Module
alias:         a simplest module
vermagic:      2.6.15.5 SMP preempt PENTIUM4 gcc-3.2
depends:

```

从中可以看出，其 `vermagic` 为“2.6.15.5 SMP preempt PENTIUM4 gcc-3.2”，运行“`insmod hello.ko`”命令，得到如下错误：

```

insmod: error inserting 'hello.ko': -1 Invalid module format
hello: version magic '2.6.15.5 SMP preempt PENTIUM4 gcc-3.2' should be '2.6.15.5 686 gcc-3.2'

```

原因在于加载该 `hello.ko` 时候所使用的内核虽然还是 Linux 2.6.15.5，但是和编译 `hello.ko` 时的内核的关键部分配置不一样，导致 `vermagic` 不一致，发生冲突，从而加载失败。

3. 内核模块的加载与卸载函数

在 Linux 2.6 内核中，内核模块必须调用宏 `module_init` 与 `module_exit()` 去注册初始化与退出函数。在 Linux 2.4 内核中，如果加载函数命名为 `init_module()`，卸载函数命名为 `cleanup_module()`，可以不必使用 `module_init` 与 `module_exit` 宏。因此，若使用 `module_init` 与 `module_exit` 宏，代码在 Linux 2.4 内核与 Linux 2.6 内核中都能工作，如代码清单 23.8 所示。

代码清单 23.8 同时支持 Linux 2.4/2.6 的内核模块加载/卸载函数

```

1 static int mod_init_func(void)
2 {
3     ...
4     return 0;
5 }
6
7 static void mod_exit_func(void)
8 {
9     ...
10 }
11
12 module_init(mod_init_func);
13 module_exit(mod_exit_func);

```



4. 内核模块使用计数

不管是在 Linux 2.4 内核还是在 Linux 2.6 内核中, 当内核模块正在被使用时, 是不允许被卸载的, 内核模板使用计数用来反映模块的使用情况。Linux 2.4 内核中, 模块自身会通过 MOD_INC_USE_COUNT、MOD_DEC_USE_COUNT 宏来管理自己被使用的计数。Linux 2.6 内核提供了更健壮、灵活的模块计数管理接口 try_module_get(&module)及 module_put (&module) 取代 Linux 2.4 中的模块使用计数管理宏。而且, Linux 2.6 内核下, 对于为具体设备写驱动的开发人员而言, 基本无须使用 try_module_get()与 module_put(), 设备驱动框架结构中的驱动核心往往已经承担了此项工作。

5. 内核模块导出符号

在 Linux 2.4 内核下, 默认情况下模块中的非静态全局变量及函数在模块加载后会输出到内核空间。而在 Linux 2.6 内核下, 默认情况时模块中的非静态全局变量及函数在模块加载后不会输出到内核空间, 需要显式调用宏 EXPORT_SYMBOL 才能输出。所以在 Linux 2.6 内核的模块下, EXPORT_NO_SYMBOLS 宏的调用没有意义, 是空操作。在同时支持 Linux 2.4 内核与 Linux 2.6 内核的设备驱动中, 可以通过代码清单 23.9 来导出模块的内核符号。

代码清单 23.9 同时支持 Linux 2.4/2.6 内核的导出内核符号代码段

```
1 #include <linux/module.h>
2 #ifndef LINUX26
3     EXPORT_NO_SYMBOLS;
4 #endif
5 EXPORT_SYMBOL(var);
6 EXPORT_SYMBOL(func);
```

另外, 如果需要在 Linux 2.4 内核下使用 EXPORT_SYMBOL, 必须在 CFLAGS 中定义 EXPORT_SYMTAB, 否则编译将会失败。

从良好的代码风格角度出发, 模块中不需要输出到内核空间且不需为模块中其他文件所用的全局变量及函数最好显式申明为 static 类型, 需要输出的内核符号最好以模块名为前缀。模块加载后, Linux 2.4 内核下可通过/proc/ksyms, Linux 2.6 内核下可通过/proc/kallsyms 查看模块输出的内核符号。

6. 内核模块输入参数

在 Linux 2.4 内核下, 通过 MODULE_PARM(var,type)宏来向模块传递命令行参数。var 为接受参数值的变量名, type 为采取如下格式的字符串[min[-max]][{b,h,i,l,s}。min 及 max 用于表示当参数为数组类型时, 允许输入的数组元素的个数范围; b 指 byte, h 指 short, i 指 int, l 指 long, s 指 string。

在 Linux 2.6 内核下, 宏 MODULE_PARM(var,type)不再被支持, 而是使用 module_param(name, type, perm)和 module_param_array(name, type, nump, perm)宏。

同样地, 为了使驱动能根据内核的版本分别调用不同的宏导出内核符号, 可以使用类似代码清单 23.10 所示的方法。

代码清单 23.10 同时支持 Linux 2.4/2.6 的模块输入参数范例

```
1 #include <linux/module.h>
2 #ifdef LINUX26
```

```

3  #include <linux/moduleparam.h>
4  #endif
5  int int_param = 0;
6  char *string_param = "I love Linux";
7  int array_param[4] =
8  {
9      1, 1, 1, 1
10 };
11 #ifdef LINUX26
12     int len = 1;
13 #endif
14 #ifdef LINUX26
15     MODULE_PARM(int_param, "i");
16     MODULE_PARM(string_param, "s");
17     MODULE_PARM(array_param, "1-4i");
18 #else
19     module_param(int_param, int, 0644);
20     module_param(string_param, charp, 0644);
21     #if LINUX_VERSION_CODE >=
22         KERNEL_VERSION(2, 6, 10)
23         module_param_array(array_param, int,
24             &len, 0644);
25     #else
26         module_param_array(array_param, int, len, 0644);
27     #endif
28 #endif

```

7. 内核模块别名、加载接口

Linux 2.6 内核在 `linux/module.h` 中提供了 `MODULE_ALIAS(alias)` 宏，模块可以通过调用此宏为自己定义一个或若干个别名。而在 Linux 2.4 内核下，用户只能在 `/etc/modules.conf` 中为模块定义别名。

加载内核模块的接口 `request_module()` 在 Linux 2.4 内核下为 `request_module(const char *module_name)`，在 Linux 2.6 内核下则为 `request_module(const char *fmt, ...)`。在 Linux 2.6 内核下，驱动开发人员可以通过调用以下的方法来加载内核模块。

```

request_module("xxx");
request_module("char-major-%d-%d", MAJOR(dev), MINOR(dev));

```

8. 结构体初始化

在 Linux 2.4 内核中，习惯以代码清单 23.11 所示的方法来初始化结构体，即“成员:值”的方式。

代码清单 23.11 Linux 2.4 内核中结构体初始化习惯

```

1 static struct file_operations lp_fops =
2 {
3     owner: THIS_MODULE,
4     write: lp_write,
5     ioctl: lp_ioctl,
6     open: lp_open,
7     release: lp_release,
8 };

```

但是，在 Linux 2.6 内核中，为了尽量向标准 C 靠拢，习惯使用如代码清单 23.12 所示的方法



来初始化结构体, 即“成员=值”的方式。

代码清单 23.12 Linux 2.6 内核中结构体初始化习惯

```
1 static struct file_operations lp_fops =
2 {
3     .owner      = THIS_MODULE,
4     .write      = lp_write,
5     .ioctl      = lp_ioctl,
6     .open       = lp_open,
7     .release    = lp_release,
8 };
```

9. 字符设备驱动

在 Linux 2.6 内核中, 将 Linux 2.4 内核中都为 8 位的主次设备号分别扩展为 12 位和 20 位。鉴于此, Linux 2.4 内核中的 `kdev_t` 被废除, Linux 2.6 内核中新增的 `dev_t` 拓展到了 32 位。在 Linux 2.4 内核中, 通过 `inode->i_rdev` 即可得到设备号, 而在 Linux 2.6 内核中, 为了增强代码的可移植性, 内核中新增了 `iminor()` 和 `imajor()` 这两个函数来从 `inode` 获得设备号。

在 Linux 2.6 内核中, 对于字符设备驱动, 提供了专门用于申请/动态分配设备号的 `register_chrdev_region()` 函数和 `alloc_chrdev_region()` 函数, 而在 Linux 2.4 内核中, 对设备号的申请和注册字符设备的行为都是在 `register_chrdev()` 函数中进行的, 没有单独的 `cdev` 结构体, 因此也不存在 `cdev_init()`、`cdev_add()`、`cdev_del()` 这些函数。要注意的是, `register_chrdev()` 在 Linux 2.6 内核中仍然被支持, 但是不能访问超过 256 的设备号。

其次, `devfs` 设备文件系统在 Linux 2.6 内核中被取消了, 因此, 最新的驱动中也不宜再调用 `devfs_register()`、`devfs_unregister()` 这样的函数。

● proc 操作。

以前的 `/proc` 中只能给出字符串, 而新增的 `seq_file` 操作使得 `/proc` 中的文件能导出如 `long` 等多种数据, 为了支持这一新的特性, 需要实现 `seq_operations` 结构体中的 `seq_printf()`、`seq_putc()`、`seq_puts()`、`seq_escape()`、`seq_path()`、`seq_open()` 等成员函数。

● 内存分配。

Linux 2.4 和 Linux 2.6 在内存分配方面发生了一些细微的变化, 这些变化主要包括:

<linux/malloc.h> 头文件被改为 <linux/slab.h>;

分配标志 `GFP_BUFFER` 被 `GFP_NOIO` 和 `GFP_NOFS` 取代;

新增了 `__GFP_REPEAT`、`__GFP_NOFAIL` 和 `__GFP_NORETRY` 分配标志;

页面分配函数 `alloc_pages()`、`get_free_page()` 被包含在 <linux/gfp.h> 中;

对 NUMA 系统新增了 `alloc_pages_node()`、`free_hot_page()`、`free_cold_page()` 函数;

新增了内存池;

针对 `r-cpu` 变量的 `DEFINE_PER_CPU()`、`EXPORT_PER_CPU_SYMBOL()`、`EXPORT_PER_CPU_SYMBOL_GPL()`、`DECLARE_PER_CPU()`、`DEFINE_PER_CPU()` 等宏因为抢占调度的出现而变得不安全, 被 `get_cpu_var()`、`put_cpu_var()`、`alloc_percpu()`、`free_percpu()`、`per_cpu_ptr()`、`get_cpu_ptr()`、`put_cpu_ptr()` 等函数替换。

● 内核时间变化。

在 Linux 2.6 中, 一些平台的节拍 (Hz) 发生了变化, 因此引入了新的 64 位计数器 `jiffies_64`,

新的时间结构体 `timespec` 增加了 `ns` 成员变量, 新增了 `add_timer_on()` 定时器函数, 新增了 `ns` 级延时函数 `ndelay()`。

- 并发/同步。

任务队列 (task queue) 接口函数都被取消, 新增了 `work queue` 接口函数。

- 音频设备驱动。

Linux 2.4 内核中音频设备驱动的默认框架是 OSS, 而 Linux 2.6 内核中音频设备驱动的默认框架则是 ALSA, 这显示 ALSA 是一种未来的趋势。

在内核的更新过程中, 大部分驱动源代码也随着内核中的 API 变更而修改了, 如下面的列表分别摘录了 `linux-2.4.18` 和 `linux-2.6.15.5` 中的并口打印机字符设备驱动 `drivers/char/lp.c` 的源代码:

<pre>static struct file_operations lp_fops = { .owner: THIS_MODULE, .write: lp_write, .ioctl: lp_ioctl, .open: lp_open, .release: lp_release, #ifdef CONFIG_PARPORT_1284 .read: lp_read, #endif }; MODULE_PARAM(parport, "l-" _MODULE_ STRING(LP_NO) "s"); MODULE_PARAM(reset, "i"); static int lp_release(struct inode *inode, struct file *file) { unsigned int minor = MINOR(inode ->i_rdev); lp_claim_parport_or_block (&lp_table[minor]); parport_negotiate(lp_table[minor].dev ->port, IEEE1284_MODE_COMPAT); lp_table[minor].current_mode = IEEE1284_MODE_COMPAT; lp_release_parport(&lp_table[minor]); lock_kernel(); kfree(lp_table[minor].lp_buffer); lp_table[minor].lp_buffer = NULL; LP_F(minor) &= ~LP_BUSY; unlock_kernel(); return 0; }</pre>	<pre>static struct file_operations lp_fops = { .owner = THIS_MODULE, .write = lp_write, .ioctl = lp_ioctl, .open = lp_open, .release = lp_release, #ifdef CONFIG_PARPORT_1284 .read = lp_read, #endif }; module_param_array(parport, charp, NULL, 0); module_param(reset, bool, 0); static int lp_release(struct inode *inode, struct file *file) { unsigned int minor = iminor(inode); lp_claim_parport_or_block (&lp_table[minor]); parport_negotiate(lp_table[minor].dev ->port, IEEE1284_MODE_COMPAT); lp_table[minor].current_mode = IEEE1284_MODE_COMPAT; lp_release_parport(&lp_table[minor]); kfree(lp_table[minor].lp_buffer); lp_table[minor].lp_buffer = NULL; LP_F(minor) &= ~LP_BUSY; return 0; }</pre>
--	--

如果驱动源代码要同时支持 Linux 2.4 和 Linux 2.6 内核, 其实也非常简单, 因为通过 `linux/version.h` 中的 `LINUX_VERSION_CODE` 可以获知内核版本, 之后便可以针对不同的宏定义实现不同的驱动源代码, 如代码清单 23.13 所示。



代码清单 23.13 同时支持 Linux 2.4 和 Linux 2.6 内核的驱动编写方法

```
1 #include <linux/version.h>
2 #if LINUX_VERSION_CODE >= KERNEL_VERSION(2, 6, 0)
3 #define LINUX26
4 #endif
5 #ifdef LINUX26
6 /*Linux 2.6 内核中的代码*/
7 #else
8 /*Linux 2.4 内核中的代码 */
9 #endif
```



除了 Linux 2.4 和 Linux2.6 之间内核的变更较大以外, Linux 2.6 的各个小版本向前推进的时候, 驱动的架构也可能发生较大的变动, 可以这么说, 几乎在不断变动中, 这是 Linux 官方内核演进的特点。因此, 运行于 2.6.x 的驱动不一定能运行于 2.6.y, 这些时候, 我们要注意这两个版本之前的差异, 并进行相应的修改。

23.4 Linux 与其他操作系统之间的驱动移植

在公司的项目更替过程中, 可能会出现操作系统的更换, 譬如类似的产品中, 以前用 VxWorks 或 Windows CE, 现在想改用 Linux。以前同类产品的 VxWorks/WinCE 运行稳定, 且设备驱动也被经过严格测试, 现在要换成 Linux 了, 是不是 VxWorks/WinCE 下的工作就完全作废了呢?

在采用原操作系统的系统中, 底层的硬件操作代码已经经过验证, 这一部分可以被 Linux 利用; VxWorks、WindowsCE 等操作系统驱动的架构和 Linux 的驱动架构有一定的相似性, 经过适当的修改, 就可以被移植到 Linux 中。

本节将以 VxWorks 为例讲解 VxWorks 驱动向 Linux 驱动的移植方法。代码清单 23.14 所示为 VxWorks 下的 LPT 并口这一典型的字符设备驱动的骨干。

代码清单 23.14 VxWorks 下的 LPT 驱动

```
1 LOCAL int lptOpen(LPT_DEV *pDev, char *name, int mode);
2 LOCAL int lptRead(LPT_DEV *pDev, char *pBuf, int size);
3 LOCAL int lptWrite(LPT_DEV *pDev, char *pBuf, int size);
4 LOCAL STATUS lptIoctl(LPT_DEV *pDev, int function, int arg);
5 LOCAL void lptIntr(LPT_DEV *pDev);
6 LOCAL void lptInit(LPT_DEV *pDev);
7
8 /*初始化设备驱动*/
9 STATUS lptDrv(int channels, /* LPT 通道 */
10 LPT_RESOURCE *pResource /* LPT 资源 */
11 )
12 {
13     int ix;
14     LPT_DEV *pDev;
15
16     /* 检查驱动是否已被安装 */
17     if (lptDrvNum > 0)
18         return (OK);
```

```

19
20 if (channels > N_LPT_CHANNELS)
21     return (ERROR);
22
23 for (ix = 0; ix < channels; ix++, pResource++)
24 {
25     pDev = &lptDev[ix];
26
27     pDev->created = FALSE;
28     pDev->autofeed = pResource->autofeed;
29     pDev->inservice = FALSE;
30
31     if (pResource->regDelta == 0)
32         pResource->regDelta = 1;
33
34     pDev->data = LPT_DATA_RES(pResource);
35     pDev->stat = LPT_STAT_RES(pResource);
36     pDev->ctrl = LPT_CTRL_RES(pResource);
37     pDev->intCnt = 0;
38     pDev->retryCnt = pResource->retryCnt;
39     pDev->busyWait = pResource->busyWait;
40     pDev->strobeWait = pResource->strobeWait;
41     pDev->timeout = pResource->timeout;
42     pDev->intLevel = pResource->intLevel;
43
44     /* 创建二进制信号量 */
45     pDev->syncSem = semBCreate(SEM_Q_FIFO, SEM_EMPTY);
46     /* 创建互斥信号量 */
47     pDev->muteSem = semMCreate(SEM_Q_PRIORITY | SEM_DELETE_SAFE
48         | SEM_INVERSION_SAFE);
49     /* 连接中断 */
50     (void)intConnect((VOIDFUNCPTR*) INUM_TO_IVEC(pResource->intVector),
51         (VOIDFUNCPTR) lptIntr, (int)pDev);
52
53     sysIntEnablePIC(pDev->intLevel); /* 开中断 */
54
55     lptInit(&lptDev[ix]);
56 }
57
58 /* 注册驱动 */
59 lptDrvNum = iosDrvInstall(lptOpen, (FUNCPTR) NULL, lptOpen,
60     (FUNCPTR) NULL, lptRead, lptWrite, lptIoctl);
61
62 return (lptDrvNum == ERROR ? ERROR : OK);
63 }
64
65
66 /* lptOpen - 打开 LPT */
67 LOCAL int lptOpen(LPT_DEV *pDev, char *name, int mode)
68 {
69     return ((int)pDev);
70 }
71
72 /* lptRead - 读并口 */
73 LOCAL int lptRead(LPT_DEV *pDev, char *pBuf, int size)

```



```
74 {
75     return (ERROR);
76 }
77
78 /* lptWrite - 写并口
79  * 返回值: 被写入的字节数, 或者 ERROR
80  */
81 LOCAL int lptWrite(LPT_DEV *pDev, char *pBuf, int size)
82 {
83     int byteCnt = 0;
84     BOOL giveup = FALSE;
85     int retry;
86     int wait;
87
88     if (size == 0)
89         return (size);
90
91     semTake(pDev->muteSem, WAIT_FOREVER); //获取互斥
92
93     retry = 0;
94     while ((sysInByte(pDev->stat) &S_MASK) != (S_SELECT | S_NODERR | S_NOBUSY))
95     {
96         if (retry++ > pDev->retryCnt)
97         {
98             if (giveup)
99             {
100                 errnoSet(S_ioLib_DEVICE_ERROR);
101                 semGive(pDev->muteSem);
102                 return (ERROR);
103             }
104             else
105             {
106                 lptInit(pDev);
107                 giveup = TRUE;
108             }
109         }
110         wait = 0;
111         while (wait != pDev->busyWait)
112             wait++;
113     }
114
115     retry = 0;
116     do
117     {
118         wait = 0;
119         sysOutByte(pDev->data, *pBuf);
120         while (wait != pDev->strobeWait) wait++;
121         sysOutByte(pDev->ctrl, sysInByte(pDev->ctrl) | C_STROBE | C_ENABLE);
122         while (wait)
123             wait--;
124         sysOutByte(pDev->ctrl, sysInByte(pDev->ctrl) &~C_STROBE);
125
126         if (semTake(pDev->syncSem, pDev->timeout * sysClkRateGet()) == ERROR)
127         {
128             if (retry++ > pDev->retryCnt)
```

```

129     {
130         errnoSet(S_ioLib_DEVICE_ERROR);
131         semGive(pDev->muteSem); //释放互斥
132         return (ERROR);
133     }
134 }
135 else
136 {
137     pBuf++;
138     byteCnt++;
139 }
140 }while (byteCnt < size) ;
141
142 semGive(pDev->muteSem); //释放互斥
143
144 return (size);
145 }
146
147 /* lptIoctl - 设备特定的控制 */
148 LOCAL STATUS lptIoctl(LPT_DEV *pDev, /* 要控制的设备 */
149 int function, /* 命令 */
150 int arg /* 参数 */
151 )
152 {
153     int status = OK;
154
155     semTake(pDev->muteSem, WAIT_FOREVER);
156     switch (function)
157     {
158         case LPT_SETCONTROL:
159             sysOutByte(pDev->ctrl, arg);
160             break;
161
162         case LPT_GETSTATUS:
163             *(int*)arg = sysInByte(pDev->stat);
164             break;
165
166         default:
167             (void)errnoSet(S_ioLib_UNKNOWN_REQUEST);
168             status = ERROR;
169             break;
170     }
171     semGive(pDev->muteSem);
172
173     return (status);
174 }
175
176 /* lptIntr - 中断处理函数 */
177 LOCAL void lptIntr(LPT_DEV *pDev)
178 {
179     pDev->inservice = TRUE;
180     pDev->intCnt++;
181     semGive(pDev->syncSem); //释放同步信号量
182     pDev->inservice = FALSE;
183     sysOutByte(pDev->ctrl, sysInByte(pDev->ctrl) &~C_ENABLE);

```



```
184 }
185
186 /* lptInit - 初始化 LPT 端口 */
187 LOCAL void lptInit(LPT_DEV *pDev)
188 {
189     sysOutByte(pDev->ctrl, 0); /* init */
190     taskDelay(sysClkRateGet() >> 3); /* hold min 50 mili sec */
191     if (pDev->autofeed)
192         sysOutByte(pDev->ctrl, C_NOINIT | C_SELECT | C_AUTOFEED);
193     else
194         sysOutByte(pDev->ctrl, C_NOINIT | C_SELECT);
195 }
```

上述驱动和 Linux 下的字符设备驱动很多相似之处，有如下体现。

- 同样包含了 `open()`、`read()`、`write()`、`ioctl()` 等函数，而且同样要注册驱动（使用 `iosDrvInstall()`）；
- 同样包含中断号和中断处理函数的绑定过程，只是用 `intConnect()`，而不是 `request_irq()`；
- 为了避免并发和竞争或进行同步，同样定义了信号量和互斥，只是信号量和互斥的名字发生了变化，而且用 `semBCreate()`、`semMCreate()` 这样的函数来创建。在访问临界资源时，也一样用互斥加以保护。

两者的差异如下。

- 内核对象、API 的名称和参数有很大不同，VxWorks 与 Linux 对等地位的函数的返回值也不相同。表 23.2 所示为 VxWorks 和 Linux 在同步、互斥和中断方面的对比。

表 23.2 VxWorks 和 Linux 同步/互斥/中断对比

事 项	VxWorks	Linux
临界资源保护	互斥（二进制信号量）	自旋锁、信号量
中断	没有顶底半部的概念，但是中断服务程序也通过 <code>semGive()</code> 和 <code>MsgQSend()</code> 等方式唤醒任务去完成中断引发的事务	有顶底半部的概念，顶半部通过软中断、 <code>tasklet</code> 、 <code>workqueue</code> 触发底半部，也可以只是 <code>wake_up</code> 一个进程
同步	同步 <code>semGive()</code> 、 <code>MsgQSend()</code> 进行	通过等待队列、完成量进行

- VxWorks 中的命名习惯和 Linux 系统不同，VxWorks 常用“xxxYyyZzz”命名方式，而 Linux 常用“xxx_yyy_zzz”命名规则。
- VxWorks 对外设的访问中不会出现类似 `ioremap()` 这样的物理地址到虚拟地址的映射过程。尽管 VxWorks 也能支持带 MMU 的处理器，但是对系统中的所有任务而言，其进行的是完全相同的物理地址到虚拟地址映射。

因此，在移植 VxWorks 驱动到 Linux 驱动的过程中，可以借鉴其流程，但是必须要替换 API、函数和变量命名等。

除了简单的字符设备外，VxWorks 下的 TTY 设备、MTD 设备、块设备等驱动的架构都和 Linux 下的架构有一定相似性，但是，由于 VxWorks 是一种完全定位于嵌入式系统的轻量级操作系统，因此，总体而言，VxWorks 设备驱动的架构要比 Linux 架构简单。由于 VxWorks 中并不存在内核空间和用户空间的界限，很多时候，驱动甚至可以遵循架构，直接给应用程序提供接口。

接下来分析几个典型的 VxWorks 设备驱动的框架结构。

1. 串行设备

VxWorks 下串行设备驱动的结构如图 23.1 所示, I/O 系统 (VxWorks 的 I/O 系统类似于 Linux 的 VFS, 都是向用户提供统一的文件操作接口 `open()`、`write()`、`read()`、`ioctl()`、`close()` 等) 不直接与串行设备驱动打交道, 而是通过 `ttyDrv` 进行。

`ttyDrv` 是一个虚拟的设备驱动, 它与 `tylib` 一起, 用于处理 I/O 系统与底层实际设备之间的通信。它们完成的工作包括在驱动表中添加相应的驱动条目、创建设备标识符, 实现与上层标准 I/O 函数及实际驱动程序连接, `ttyDrv` 完成 `open()` 和 `ioctl()` 两项功能 (对应函数 `ttyopen()` 和 `ttyioctl()`), `tylib` 完成 `read()` 和 `write()` 两项功能 (对应函数 `tyRead()` 和 `tyWrite()`) 并管理输入/输出数据缓冲区。

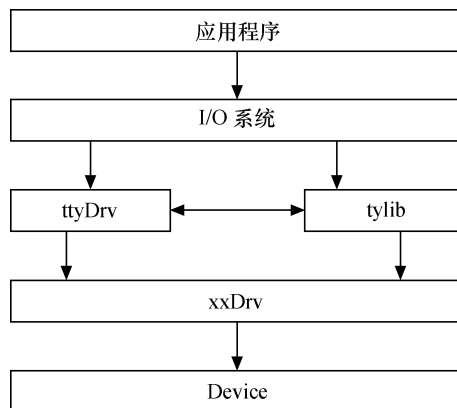


图 23.1 VxWorks 串行设备驱动的结构

在上下层数据传递方面, VxWorks 使用如图 23.2 所示的结构 (假设串口 UART 为 8250)。

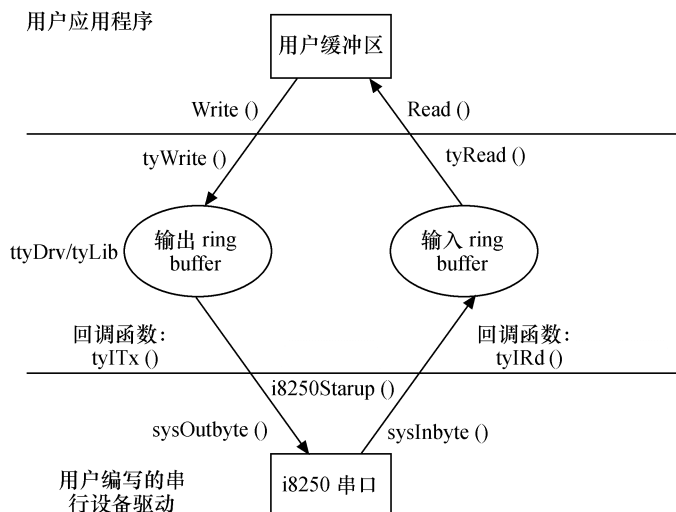


图 23.2 VxWorks 串行设备驱动数据流

当用户调用 `write()` 函数进行写操作时, 系统根据相应的文件描述符 `fd` 调用驱动表中注册的 `tyWrite()` 函数, 此函数会将用户缓冲区的内容写入相应的输出 ring buffer。当发现缓冲区内有内容时, 函数 `tyITx()` 被调用, 从 ring buffer 读取字符, 将字符发往指定的串口。

当串口接收到数据时会调用输入中断服务程序, 将输入的字符写入指定的缓冲区。然后由回调函数 `tyIRd()` 将缓冲区的内容读入 ring buffer, 当用户调用 `read()` 函数进行写操作时, 会根据文件描述符 `fd` 调用在驱动表中注册的函数 `tyRead()`, 此函数会将 ring buffer 中的内容读入用户缓冲区。

2. 块设备

如图 23.3 所示, 与 Linux 类似, VxWorks 中的块设备驱动程序也不与 I/O 系统直接作用, 而是通过磁盘、Flash 文件系统与 I/O 系统交互。文件系统把自己作为一个驱动程序装载到 I/O 系统



中, 并把请求转发给实际的设备驱动程序。块设备的驱动程序不使用 `iosDrvInstll()` 来安装驱动程序, 而是通过初始化块设备描述结构 `BLK_DEV` 或顺序设备描述结构 `SEQ_DEV`, 来实现驱动程序提供给文件系统的功能。块设备也不调用非块设备的安装函数 `iosDevAdd()`, 而是调用文件系统的设备初始化函数, 如 `dosFsDevlnit()` 等。

3. 网络设备

在 VxWorks 中, 网卡驱动程序分为 END (Enhanced Network Driver, 增强型网络驱动) 和 BSD 两种。

如图 23.4 所示, END 驱动程序基于 MUX 模式, 网络驱动程序被划分为协议组件和硬件组件。MUX 是数据链路层和网络层之间的接口, 它管理网络协议接口和底层硬件接口之间的交互。

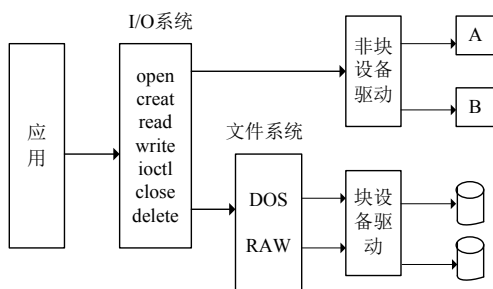


图 23.3 VxWorks 块设备驱动与 I/O 系统

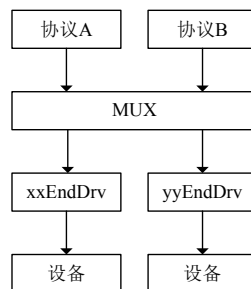


图 23.4 VxWorks 下网络设备 END 驱动

如图 23.5 所示, MUX 数据包采用 `mBlk-clBlk-cluster` 数据结构处理网络协议栈传输的数据。其中, `cluster` 保存的是实际的数据, `mBlk` 和 `clBlk` 中保存的信息用来管理 `cluster` 中保存的数据。

在数据接收过程中, 设备会直接将接收到的数据包放入内存池预先分配的 `cluster` 中并产生一个中断。如果设备不能完成上述功能, END 驱动函数应该完成将数据从 `buffer` 到 `cluster` 中的拷贝。数据被放到 `cluster` 以后, 驱动程序将通过对 `netBuflib()` 中函数的调用来完成 `mBlk-clBlk-cluster` 链的创建, 从而为数据在网络协议各层之间的传递做好准备, 创建此结构链一般需要以下 4 步。

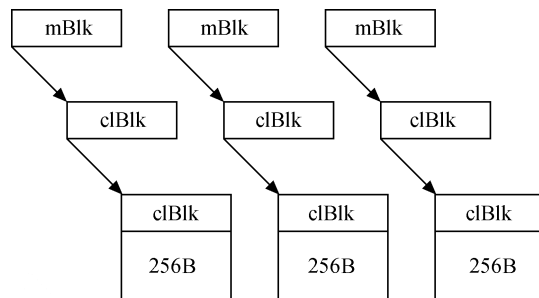


图 23.5 VxWorks 下网络设备驱动的 `mBlk-clBlk-cluster` 结构

- (1) 调用函数 `netClblkGet()`, 从内存池中取 `cluster` 结构。
- (2) 调用函数 `netClblkJoin()`, 将 `clBlk` 与存有数据的 `cluster` 连接起来。
- (3) 调用函数 `netMblkGet()`, 从内存池中取 `mBlk` 结构。
- (4) 调用函数 `netMblkClJoin()`, 将 `mBlk` 与 `clBlk-cluster` 结构连接起来。

END 设备驱动的数据包接收过程的整个流程如图 23.6 所示, 经历了“设备中断服务程序→调用 `netjobAdd()` 添加网络任务（类似于 Linux 中的底半部, 引发“底半部” `xxReceive` 被执行）→到达 MUX 层→协议层→用户 `read()` 函数”的过程。

数据包的发送是数据包接收的反过程，应用程序通过 `write()` 函数调用将要发送的数据放入应用程序数据缓冲区中后，网络协议负责将 `buffer` 中的数据放入为其分配的内存池中，并以 `mBlk-clBlk-cluster` 链的形式来存储，这样实现了向下层协议传递数据报时，传递的只是指向此数据链结构的指针，而代替了数据在各层协议之间的拷贝。

当有数据要发送时，网络协议层通过其与 MUX 层的接口调用 `muxSend()` 函数，而 `muxSend()` 函数又调用 `xxSend()` 函数将传递来的数据包送到发送 FIFO 队列中，然后启动网卡设备的发送功能，发送完后将随之产生中断信号，调用中断服务程序，清除设备缓冲区。

与 VxWorks 相比，Linux 网络设备驱动中没有 `mBlk-clBlk-cluster`，贯穿始终的是 `sk_buff`，作为数据包的“容器”，其地位与 `mBlk-clBlk-cluster` 相当。

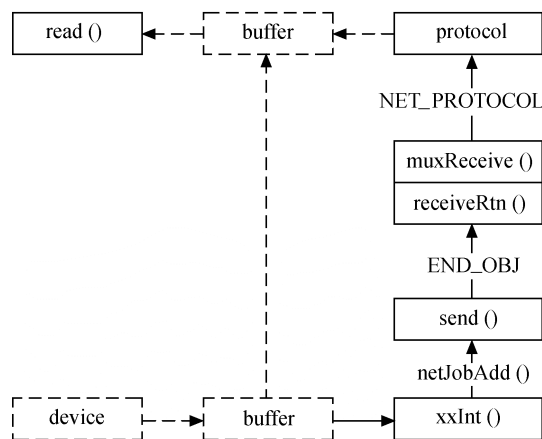


图 23.6 VxWorks 网络设备驱动的数据包接收流程

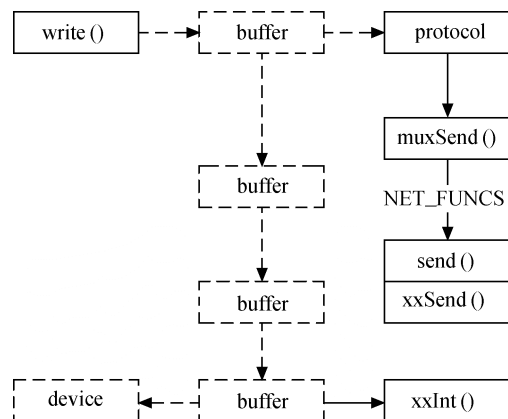


图 23.7 VxWorks 网络设备驱动的数据包发送流程

4. PCI 设备

在 VxWorks 中，PCI 设备驱动和 Linux 系统类似，PCI 部分只是一个“外壳”，设备本身所属类型的驱动才是工作的主体。与 Linux 系统类似，在 VxWorks 中，初始化 PCI 设备需要如下几个步骤。

(1) 利用供应商和设备标识确定设备的总线号、设备号和功能号，在系统中找到设备，要用的 API 是包括 `pciFindDevice()`、`pciFindClass()`，前者根据 ID、总线编号、设备编号、功能编号找到一个特定的 PCI 设备，后者寻找与参数中类匹配的 PCI 设备。

(2) 进行 PCI 设备的配置，和 Linux 系统非常相似，使用的 API 包括 `pciConfigInByte()`、`pciConfigInWord()`、`pciConfigInLong()`、`pciConfigOutByte()`、`pciConfigOutWord()`、`pciConfigOutLong()`、`pciConfigModifyLong()`、`pciConfigModifyWord()` 和 `pciConfigModifyByte()` 等。

(3) 确定映射到系统中的设备的基地址，即分析 PCI 设备的 I/O 内存/端口资源。

(4) 挂接 PCI 设备中断。

5. USB 设备

如图 23.8 所示，与 Linux 系统相似，VxWorks 中的 USB 驱动也分成了主机控制器驱动 (HCD)、USB 核心驱动层 (USBBD) 和 USB 设备驱动 (Client Driver) 几个层次。

Client Driver 负责管理连接到 USB 上的不同设备，它通过 IRP (I/O 请求包，与 Linux 系统中的 URB 类似) 向 USBBD 层发出数据接收或发送报文，它向应用层提供 API 函数，屏蔽 USB 实现



的细节, 实现数据的透明传输。

USB 通过 IRP 得到此设备的属性和本次数据通信的要求, 将 IRP 转换成 USB 所能辨识的一系列事务处理, 交给 HCD 层或者直接交给 HCD。此外, USB 还负责新设备的配置、被拔掉设备资源的释放和 Client Driver 的装载/卸载。

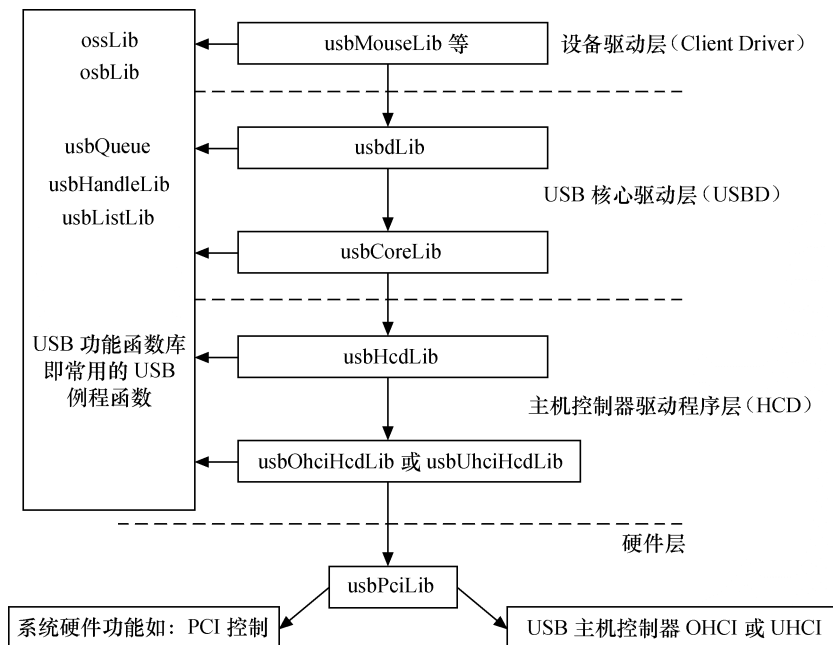


图 23.8 VxWorks 下的 USB 驱动层次

HCD 完成对 Host 控制器的管理、带宽分配、链表管理及根 Hub 功能等。它将数据按传输类型组成不同的链表, 并定义不同类型传输在一帧中所占带宽的比例, 交给 Host 控制器处理, 控制器根据规则从链表上摘下数据块, 为其创建一个或多个事务, 完成与设备的数据传输。当事务处理完成时, HCD 将结果交给 USBD 层, 由它通知对 Client Driver 进行处理。

从以上分析可知, 对于块设备、网络设备、PCI 设备、USB 设备等复杂设备而言, VxWorks 和 Linux 驱动的框架呈现出了较大的差异, 但其中也不乏共同的设计思想。Linux 驱动工程师应该在了解这些差异的情况下, 尽可能地利用 VxWorks 中现成的稳定的代码以减少 Linux 驱动的工作量, 至少硬件控制这一部分是可以通用的。

23.5 Linux 内核的移植

Linux 内核的移植主要含义是将 Linux 内核运行于一块新的 SoC 芯片或一块新的电路板之上, 其实质含义就是建立 Linux 的板级支持包 (BSP)。BSP 的本质作用有二: 为内核的运行提供底层支撑; 屏蔽与板相关的硬件细节。对于 ARM 而言, BSP 代码位于 arch/arm/的各个 plat 和 mach 目录下, 结构如下:

```

plat-xxx
linux-2.6/arch/arm/
    plat-omap/
    plat-pxa/
    plat-s3c/
    plat-s3c24xx/
    plat-s3c64xx/
    plat-stmp3xxx/

mach-xxx
linux-2.6/arch/arm/
    mach-s3c2400/
    mach-s3c2410/
    mach-s3c2412/
    mach-s3c2440/
    mach-s3c2442/
    mach-s3c2443/
    mach-s3c24a0/
    mach-s3c6400/
    mach-s3c6410/
  
```

所有 S3C6410 板的板文件都位于 arch/arm/mach-s3c6410/，如 LDD6410 的即为 arch/arm/mach-s3c6410/mach-ldd6410.c，而所有 S3C 系列芯片 BSP 公用的部分又被提炼到 arch/arm/plat-s3c/。

这些代码完成的主要工作如下。

1. 时钟 tick (Hz) 的产生

系统节拍是 Linux 操作系统得以运行的基本条件之一，为 Linux 建立节拍只需要在硬件上指定一个定时器，并以 sys_timer 的形式对其进行封装，根据 Hz 调整定时器硬件计数器，并在定时器中断的处理函数中调用 timer_tick()。代码清单 23.15 列出了 S3C6410 处理器的系统定时器。

代码清单 23.15 S3C6410 处理器的系统定时器

```

1 struct sys_timer s3c64xx_timer = {
2     .init      = s3c64xx_timer_init,
3     .offset    = s3c2410_gettimeoffset,
4     .resume    = s3c64xx_timer_setup
5 };
6
7 static void __init s3c64xx_timer_init(void)
8 {
9     s3c64xx_timer_setup();
10    setup_irq(IRQ_TIMER4, &s3c2410_timer_irq);
11 }
12
13 static irqreturn_t
14 s3c2410_timer_interrupt(int irq, void *dev_id)
15 {
16     timer_tick();
17     return IRQ_HANDLED;
18 }
19
20 static struct irqaction s3c2410_timer_irq = {
21     .name      = "S3C2410 Timer Tick",
22     .flags     = IRQF_DISABLED | IRQF_TIMER | IRQF_IRQPOLL,
23     .handler   = s3c2410_timer_interrupt,
  
```



```
24 };
```

sys_timer 结构体的 init() 成员函数用于定时器的初始化 (设置硬件计数器以产生 Hz、中断)。由于系统以 Hz 为单位更新墙上时间, 因此 Linux 的 gettimeofday() API 如果没有 offset() 的帮助是无法达到微秒级精度的, offset() 函数实际是计算当前硬件计数值与节拍之前的差异。

2. 系统中断控制的方法

BSP 中需要将系统的所有中断以 irq_chip 结构体的形式进行组织, 并实现各中断的 mask()、unmask()、ack()、mask_ack() 方法, 代码清单 23.16 给出了 S3C6410 处理中断的部分代码片段。

代码清单 23.16 S3C6410 BSP 中断处理

```
1 static struct irq_chip s3c_irq_uart = {
2     .name      = "s3c-uart",
3     .mask      = s3c_irq_uart_mask,
4     .unmask    = s3c_irq_uart_unmask,
5     .mask_ack  = s3c_irq_uart_maskack,
6     .ack       = s3c_irq_uart_ack,
7 };
8 static void __init s3c64xx_uart_irq(struct uart_irq *uirq)
9 {
10     for (offs = 0; offs < 3; offs++) {
11         irq = uirq->base_irq + offs;
12         set_irq_chip(irq, &s3c_irq_uart);
13         set_irq_chip_data(irq, uirq);
14         set_irq_handler(irq, handle_level_irq);
15         set_irq_flags(irq, IRQF_VALID);
16     }
17     set_irq_chained_handler(uirq->parent_irq, s3c_irq_demux_uart);
18 }
19 void __init s3c64xx_init_irq(u32 vic0_valid, u32 vic1_valid)
20 {
21     set_irq_chip(irq, &s3c_irq_timer);
22     ...
23     for (uart = 0; uart < ARRAY_SIZE(uart_irqs); uart++)
24         s3c64xx_uart_irq(&uart_irqs[uart]);
25 }
```

3. GPIO、DMA、时钟资源的统一管理

在 BSP 中, 通常需要将所有 GPIO 以 gpio_chip 结构体的形式进行组织, 这个结构体中的成员函数用于设置 GPIO 的方向、读取和设置 GPIO 的电平, 如代码清单 23.17 所示。

代码清单 23.17 gpio_chip 结构体

```
1 struct gpio_chip {
2     int      (*request)(struct gpio_chip *chip,
3                         unsigned offset);
4     void     (*free)(struct gpio_chip *chip,
5                     unsigned offset);
6     int      (*direction_input)(struct gpio_chip *chip,
7                                unsigned offset);
8     int      (*get)(struct gpio_chip *chip,
9                     unsigned offset);
10    int      (*direction_output)(struct gpio_chip *chip,
11                                unsigned offset, int value);
12 }
```

```

12     void      (*set)(struct gpio_chip *chip,
13                      unsigned offset, int value);
14 };

```

Linux 会提供如下一组通用关于 GPIO 的 API:

```

int gpio_request(unsigned gpio, const char *label);
void gpio_free(unsigned gpio);
int gpio_direction_input(unsigned gpio);
int gpio_direction_output(unsigned gpio, int value);
int gpio_get_value_cansleep(unsigned gpio);

```

这样做的好处是驱动的代码完全可以以与平台无关的方式申请和使用 GPIO，而不是各自为政，通过读写寄存器来访问 GPIO，大大提高了驱动的可移植性。

同样的，BSP 也要实现针对 DMA 通用 API，如：

```

int request_dma(unsigned int chan, const char * device_id);
void free_dma(unsigned int chan);
void enable_dma(unsigned int chan);
void disable_dma(unsigned int chan);
void set_dma_mode(unsigned int chan, unsigned int mode);
void set_dma_sg(unsigned int chan, struct scatterlist *sg, int nr_sg);

```

同样的，BSP 也要实现针对时钟的通用 API，包括 clk_get()、clk_put()、clk_enable()、clk_disable()、clk_get_rate()、clk_round_rate()、clk_set_rate()、clk_get_parent()、clk_set_parent()。

4. 静态映射的 I/O 内存

在 BSP 中，可以通过 map_desc 结构体和 iotable_init() 函数提前建立某段物理地址和虚拟地址之间的静态映射，该知识点我们在第 11 章“I/O 内存静态映射”一节已经进行过讲解。

5. 设备的 I/O、中断、DMA 等资源封装平台数据

主要是 platform 信息、SPI board 信息和 I²C board 信息，这些内容我们在前面各章节已经进行过讲解，这里就不再赘述。

最后，对于一块 ARM 电路板而言，我们会将它的中断初始化、静态内存映射、系统定时器等板级信息透过 MACHINE_START 和 MACHINE_END 之间的宏绑定在一起。代码清单 23.18 给出了 LDD6410 的例子。

代码清单 23.18 LDD6410 开发板的 MACHINE_START 和 MACHINE_END

```

1 MACHINE_START(SMDK6410, "LDD6410")
2 /* Maintainer: Barry Song <21cnbao@gmail.com> */
3 .phys_io = S3C_PA_UART & 0xffff0000,
4 .io_pg_offst = (((u32)S3C_VA_UART) >> 18) & 0xfffc,
5 .boot_params = S3C64XX_PA_SDRAM + 0x100,
6
7 .init_irq     = s3c6410_init_irq,
8 .map_io      = ldd6410_map_io,
9 .init_machine= ldd6410_machine_init,
10 .timer       = &s3c64xx_timer,
11 MACHINE_END

```

从代码清单 23.18 第 1 行“MACHINE_START(SMDK6410, "LDD6410")”可以看出，LDD6410 仍然使用了 SMDK6410 的 mach ID，实际上可以透过 Linux ARM 的邮件列表获得一个新的 mach ID。

大多数工程师就职于设备提供商，因此不涉及 SoC 级的移植，也就是说芯片公司已经将系统



定时器、GPIO、DMA、时钟等都封装好了，工程师只需要进行板相关的移植。这里我们给出 Linux 板级移植的原则：切忌直接修改现有电路板的代码作为自身电路板的代码，例如，如果电路板与 SMDK6410 有差异，就直接修改 `arch/arm/mach-s3c6410/mach-smdk6410.c`，这是完全错误的，正确的方法是新建自己的板文件，将本身的设备和资源填写在新的板文件里面。

除了 SoC 芯片级和板级 Linux 移植外，移植工作量最大的是体系结构相关的 Linux 移植，例如将 Linux 移植到一个全新的 CPU 体系架构，如 TI 的 DSP 芯片。则工作量还涉及内存管理、进程调度、异常和陷阱等。

23.6 总结

在编写 Linux 设备驱动的程序，要特别注意代码的可移植性，要留意数据类型的长度、结构体的对界、CPU 大小端模式以及内存页面的大小。

为了加速驱动的开发过程，在拿到一个驱动开发任务的时候，务必搜集足够的“情报”，找到可模拟的芯片或可利用的代码，这样可以事半功倍。一般而言，demo 板的驱动、类似芯片的驱动以及无操作系统时的硬件操作代码都是可以参考的代码。

Linux 2.4 到 Linux 2.6 的改进导致驱动中发生了一些细微的变化，了解这些变化后可进行驱动的更新。除了不同版本的 Linux 以外，其他操作系统中的驱动源代码经过适当的修改也可被移植到 Linux 中。